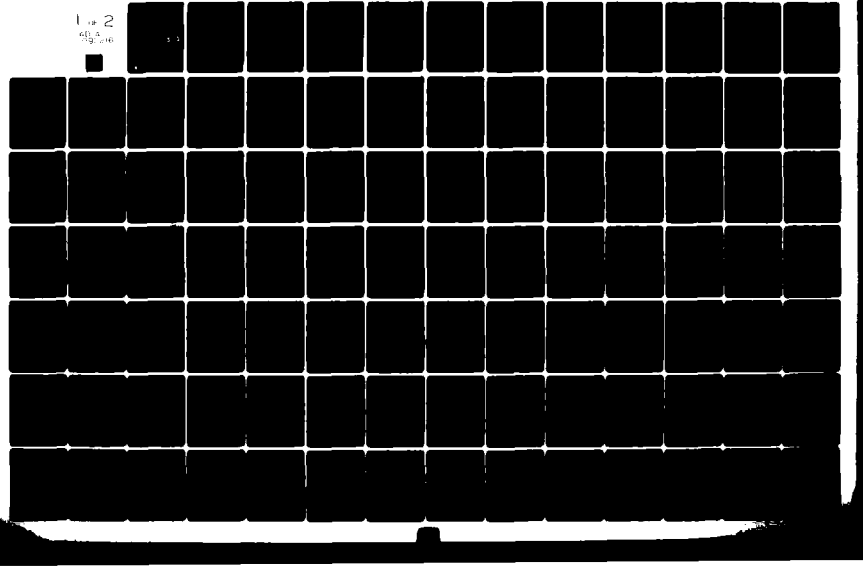
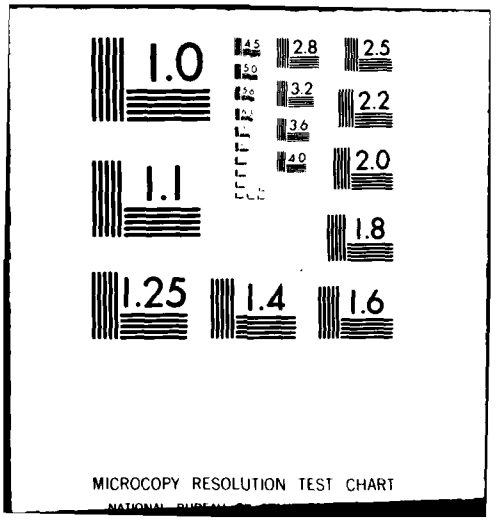
AD-A092 216 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED MULTI-MICROPROCESSOR --ETC(U)
AUG 80 R L STEWART
UNCLASSIFIED AFIT-CI-80-41T NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 80-41T	2. GOVT ACCESSION NO.: AD-A092216
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): Considerations for an Assembler Scheduled Multi-Microprocessor System,	5. TYPE OF REPORT & PERIOD COVERED: THESIS/DISSERTATION
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Richard Lee Stewart	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: Auburn University	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: master's thesis,
11. CONTROLLING OFFICE NAME AND ADDRESS: AFIT/NR, WPAFB OH 45433	12. REPORT DATE: 26 August 1980
	13. NUMBER OF PAGES: 107
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 118	15. SECURITY CLASS. (of this report): UNCLASS
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC ELECTE NOV 26 1980 S D A

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Repo

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

FREDRIC C. LYNCH, Major, USAF
Director of Public Affairs

Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

8011 24 157

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

012200

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED

MULTI-MICROPROCESSOR SYSTEM

Richard Lee Stewart

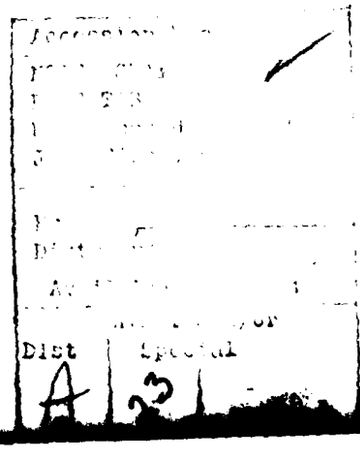
Certificate of Approval:

J. S. Boland, III
Professor
Electrical Engineering

J. R. Heath, Chairman
Assistant Professor
Electrical Engineering

V. P. Nelson
Assistant Professor
Electrical Engineering

Paul F. Parks, Dean
Graduate School



CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED MULTI-MICROPROCESSOR SYSTEM

Richard Lee Stewart

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama

August 26, 1980

CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED

MULTI-MICROPROCESSOR SYSTEM

Richard Lee Stewart

_____

Signature of Author

_____

Date

Copy sent to:

Name	Date

VITA

Captain Richard Lee Stewart, son of Harry Baker Stewart and Anne (Butler) Stewart, was born December 7, 1946, in Pittsburgh, Pennsylvania. He graduated from Hillsboro High School, Nashville, Tennessee, in 1964. In August, 1964, he entered General Motors Institute and received the degree of Bachelor of Mechanical Engineering (Automotive Design) in August, 1969. He immediately entered the United States Air Force, Officer Training School, and was commissioned a Second Lieutenant on November 13, 1969, at Lackland AFB, Texas. He received the Pilot Rating on February 12, 1971, at Vance AFB, Oklahoma. After various flying assignments he entered the Graduate School, Auburn University, in September, 1976, in the Air Force Institute of Technology, Civilian Institution Program. In March, 1978, he was assigned to the Directorate of Computer Sciences, Armament Division, Air Force Systems Command at Eglin AFB, Florida. He married Poonsri, daughter of Jang and Prem Runkshokgam of Bangkok, Thailand, in December, 1971. They have three children, Noporn Harry, Panawjidt Anne, and Wichan Richard.

THESIS ABSTRACT

CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED

MULTI-MICROPROCESSOR SYSTEM

Richard Lee Stewart

Master of Science, August 26, 1980
(B.M.E., General Motors Institute, 1969)

116 Typed Pages

Directed by J. Robert Heath

A parallel suitability or processability checker was incorporated into pass one of an INTEL 8080 cross assembler. For an assembler level source program, it yields a suitability factor for parallel processing, a jump structure analysis, and the nodes of Ramamoorthy's Loop Free Program Graph shown on the assembly language program source list. This information can be used to construct the Loop Free Program Graph.

This assembler modification is based on previous research by Ramamoorthy and others which achieved dynamic scheduling of high level language parallel processable tasks, at run time, in a multiprocessing environment. Differences in analyzing high level language (FORTRAN) and assembly language programs are explained.

Eight assembly language source programs were analyzed to test the suitability checker and investigate favorable characteristics of assembly language loops with respect to parallel processability.

Suggestions are made for further development of a parallel task recognizer for assembly language programs using Ramamoorthy's connectivity analysis method.

Design considerations are outlined for development of an assembler scheduled multi-microprocessor system. The machine would execute source program partitions in parallel on a production basis a large number of times. This would be possible after a combined assembly and schedule of the load modules.

Applications envisioned are microprocessor based controllers or instruments that would achieve increased speed at less cost by performing such operations as input, calculation, retrieval, and output simultaneously. Also, economical machines could be designed to study aspects of parallel processing for large scale computers and high level languages.

TABLE OF CONTENTS

LIST OF FIGURES ix

I. INTRODUCTION 1

II. CONSIDERATIONS AND ASSUMPTIONS 4

Favorable Characteristics of Parallel Processable Programs
Advantages of Assembly Language Suitability Checker
Assumptions for Assembly Language Suitability Checker

III. ALGORITHM FOR SUITABILITY CHECKER 16

Common Areas and Variables
Counters and Other Variables
Instruction Scan
Checking Loop Constructs
Nest Check
Final Checks
Jump Analysis
Loop Free Program Graph

IV. SUITABILITY FACTOR AND DIAGNOSTIC MESSAGES 45

Suitability Factor
Possible Errors Noted by the Analysis
Warnings Made by the Analysis
Notes Made by the Analysis

V. EXPERIMENTAL RESULTS 52

Findings
Explanation of a Sample Program Output

VI. CONCLUSIONS AND RECOMMENDATIONS 69

Significance of this Work
General Conclusions
Suggested Complementary Work
Scheduling Considerations
Configuration and Use of the Multi-Microprocessor System

BIBLIOGRAPHY 78

APPENDICES 80

A. Suggested Interprocessor Communications

B. Assembler Modifications Listing

C. Program Specifications of the Modified Assembler

D. INTEL 8080 Op-Code Groups

LIST OF FIGURES

1. Graphs Used By the Parallel Task Recognizer 5
2. FORTRAN-Like Loops in Assembly Language 13
3. Parallel Processing Within a Loop 15
4. Modifications to the Assembler Program 17
5. Analysis for a Simple Microprocessor Program 18
6. Variable Initialization 22
7. Flowchart for Instruction Scanner 24
8. Flowchart for Finding Forward Jump Destinations 26
9. Flowchart for Checking Loop Constructs 27
10. Task Convergence and Overlapped Loops 34
11. Flowchart for Nest Check 35
12. Flowchart for Final Checks and Jump Analysis 38
13. Flowchart for Subroutine LSTOUT Modification 41
14. Flowchart for Subroutine FINDLP 44
15. Length of Nested Loops 47
16. Experimental Results 53
17. Sample Program Output 55
18. Loop Free Graph of the Sample Program 67
19. PACE Instruction Decoding Sequence 76

I. INTRODUCTION

This thesis addresses the problem of how to better exploit the low cost of microprocessors and overcome the drawback of limited speed capability. The method investigated involves using several co-operating parallel processors for faster execution of a single program that will be assembled once and run many times. This would spread the optimization cost over a very large number of applications [1].

Many reasons have been given for parallel processing. A very significant increase in system throughput is theoretically possible depending on the system and the application [2-10]. Almost every computer program has some potential for parallel processing, because the input/output (I/O) can be overlapped with the function performed by the computer [11]. The relation of parallel processing to time sharing also has been discussed as justification [12]. Speed is not the only reason however.

Memory and processors can be used more effectively. Microprocessors are now very inexpensive, and using more processors better utilizes the more expensive memory. This is possible in a wide range of applications. Most microprocessor systems are interrupt driven and should have good potential for parallel processing because the interrupt task can usually be done concurrently with the main program.

While improving existing systems, the benefits can be compounded by developing design guidelines for future systems.

Using microprocessor systems would be a cost effective means for further research on parallel processing. This research is needed, because many such systems have been suggested, but few attempts have been made to partition the application programs into parallel processable segments [8,9]. Some work has been done in this area for use with large scale computers, however.

Solutions to the problems of synchronizing shared resources have been found by Djkstra, Knuth, and Coffman [11]. Previous research on parallel processability and task partitioning on high level language programs has been done by Bernstein [12] and Ramamoorthy [1,11,13-15]. How does this previous work relate to microprocessors?

Ramamoorthy's results with FORTRAN programs are applicable to a variety of uses on a limited scale [1,14]. The source program must be less than 200 executable statements. It is executed on a large scale CDC 6600 computer, and the parallel processes are scheduled dynamically during execution. Only non-nested DO loops are allowed. Little apparent interest or exploitation of these techniques was shown between 1971 and 1978. But in the past 18 months there has been increased commercial interest in multi-microprocessor systems and concurrent processing [2-9]. However, multi-microprocessor software and systems have not developed along the guidelines, suggested by Ramamoorthy, for large computers.

The currently developing distributed multi-microprocessor systems (master/slaves) do not fully utilize the master processor and are

usually uniquely specialized systems that are not generally applicable to a variety of uses. They are, therefore, sometimes not as cost effective as possible. This is parallelism at the operating system level rather than at the application program level; multiprogramming vs. parallel processing.

There is an untapped potential for a more generalized system that makes parallel processing almost transparent to the user or microprocessor system designer. The following chapters will discuss:

1. Factors bearing on the problem;
2. Partial solution - suitability checker;
3. Interpretation of results;
4. Suggestions for further work.

It is assumed that readers have a rudimentary knowledge of common terms used to describe an assembler program.

II. CONSIDERATIONS AND ASSUMPTIONS

The problem of implementing an assembler scheduled multi-microprocessor system may be broken into five parts [1,4]. First, appropriate candidate programs must be found by using a suitability checker. Secondly, parallel processable portions of the program must be identified using a parallel task recognizer. Thirdly, synchronization primitives must be added for interprocessor communications and scheduling. Fourthly, utilizing the parallel task recognizer and scheduling information, the program must be loaded into memory for parallel processing. Lastly, the memory organization and system hardware must be defined. These are significant problems for assembly language source programs because of the simplicity and fundamental nature of microprocessor based systems. Therefore, an attempt was not made to solve the entire problem. This thesis deals mainly with part one and portions of part two of the problem. This includes, for assembly language source programs, making a suitability determination and finding elements of Ramamoorthy's reduced or loop free program graph (LFPG) [13,14]. The LFPG has a node for each instruction in the source program except for the case of loops. For loops, all instructions or tasks are grouped together and represented as a single node; thus the term, loop free. Fig. 1 illustrates examples of the graphs used by Ramamoorthy's Parallel Task Recognizer. It should be emphasized that

Original Program to Add 32 Bit Numbers[1]				Analysis of Task Transitions[2]	
Task	Source Code			Function of	Action
1	.ADD 32	LXI	B, 3	immed. operand	r B = 3
2		DAD	B	task 1, r H&L	r H&L = r H&L+r B&C
3		XCHG		task 2, r D&E	exchg.r D&E, r H&L
4		DAD	B	tasks 1 & 3	r H&L = r H&L+r B&C
5		STC		nothing	set carry
6		CMC		task 5	reset carry
7	.LOOP	LDAX	D	task 3	load addend 1
8		ADC	M	tasks 4,6 & 7	add acc.+r H&L, carry
9		STAX	D	task 8	store result
10		DCX	D	task 3	r D&E = r D&E - 1
11		DCX	H	task 4	r H&L = r H&L - 1
12		DCR	C	r C	decrement loop index
13		JP	.LOOP	task 12	loop if positive
14		RET		task 13	return

Figure 1

Graphs Used by the Parallel Task Recognizer

[1]This subroutine requires that register pair H&L point to the first byte of the first number. Register pair D&E points to the first byte of the second number. Register C is set to two.

[2]Transitions exist only between tasks which change a value and the next task which uses that same value. It is not really necessary to analyze task transitions within the loop, but this is done for clarity and completeness. As is shown in the Parallel Processable Task Graph on the following page, a transition exists from task 1 to task 2, because task 2 uses the results of task 1, etc.

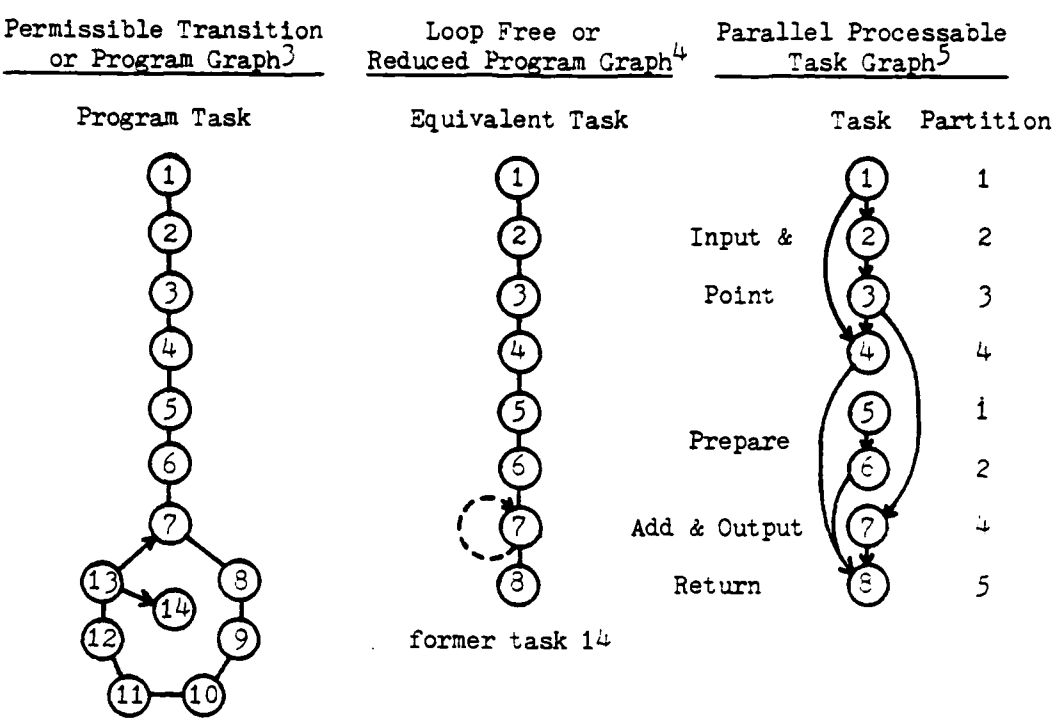


Figure 1 (continued)

Graphs Used by the Parallel Task Recognizer

[3]Each node represents a statement in the program shown on the previous page. Unless an element is a branch, its successor is the next node.

[4]Program representation in which all elements of a loop are considered as a single task or a node on the graph.

[5]Time ordering exists between nodes connected by arrows. Partitions are identified by using Ramamoorthy's Matrix Method of Precedence Partitions [14]. Partitions would be executed by two processors. Tasks in the same partition are executed concurrently. The numbers to be added and the result would be stored in common memory. The subroutine could run up to 19 percent faster, i.e., 16 tasks vs. 21 tasks for the complete process. Exactly how much faster depends on the overhead of initialization and transfer of information between the two processors. If the routine were being called from a loop, the benefits would increase with each iteration of the loop.

Ramamoorthy's work was based on high level language source programs. Here, the source programs are in microprocessor assembly language which significantly complicates the overall task. This chapter first discusses the problems of implementing a suitability checker by checking for favorable characteristics in pass one of the assembler. Second, the differences in working with FORTRAN and assembly language are addressed, because Ramamoorthy's work dealt only with FORTRAN programs.

Favorable Characteristics of Parallel Processable Programs

Bernstein stated the conditions necessary for parallel processing and why program suitability is programmer dependent when dealing with implicit (vs. explicit) parallelism [12]. The same task or algorithm could be coded more or less favorably depending on the programmer's style or sequence. Based partially on Bernstein's work, Ramamoorthy devised a hueristic formula for determining suitability for parallel processing [1].

The formula has nine variables used with FORTRAN in calculating a suitability factor, SF:

$$SF = \frac{NR + NA + NP + NC + ND - NI - NG}{LP - LD} - \frac{LD}{LP}$$

where: NR = READS or input
NA = Arithmetic statements
NP = PRINTS or output
NC = CALLS
ND = DO loops (loops of known boundaries and iterations)
NI = IF's or conditional branches
NG = GO TO's or unconditional branches
LP = Total executable statements
LD = Total statements in DO loops

This formula was based on research gathered after using Ramamoorthy's parallel task recognizer to analyze FORTRAN programs. It shows as plus factors, those tasks that could be done simultaneously given that Bernstein's conditions were satisfied. The negative factors represent conditions that delay scheduling decisions until execution time, i.e. conditional branches and unconditional forward branches. These statements can generate intricate paths which complicate prediction of process flow [12]. The factors LP and LD reflect the fact that DO loops enhance possibility for parallel processing provided the loops are not too long with respect to other partitions (tasks that can be executed as a block). Obviously, if the loops are very long, other partitions would be executed before the loop finished. One processor would have to wait so long for the loop to finish, that benefits of parallel processing would be lost. It would be better in such a case to process sequentially and not incur the overhead of establishing parallel processes, because the overhead might cancel any benefits of parallel execution.

If this works for FORTRAN it would seem to be applicable for any language. But it is significant to note that although the algorithm for a suitability checker or parallel task recognizer is language independent, its implementation is obviously dependent upon the level and structure of the source language to a great extent. Thus a completely universal application is not possible [11,14]. If further work is necessary to implement the recognizer on another language, why choose assembly language?

Advantages of Assembly Language Suitability Checker

There are many significant reasons to analyze assembly language for parallel processing. First, there is a large potential to improve many existing assembly language programs, since they are usually more memory efficient than high level language programs [3]. Second, this research could indicate the need for high level languages such as PASCAL that allow explicit indications of parallelism to be used on microprocessors [8,11]. Third, we can identify and standardize the most effective parallel constructs as desirable explicit capabilities of high level microprocessor languages [9]. Fourth, utilizing implicit parallelisms in existing assembly language does not require the programmer to learn a new language. Fifth, studying parallel processing with small computer systems would be cost effective. And when the results are better understood, the techniques may be applied to more sophisticated systems for further benefits. How then can this analysis be applied to assembly language?

Two main differences exist between FORTRAN and assembly language with respect to implementing the suitability checker and parallel task recognizer. One, INTEL 8080 language has no explicit constructs for loops of known iteration or length that can be easily determined by scanning a single line of the source program. But loops in assembly language can be compared to certain characteristics of FORTRAN-like loops, to find the loops for which the analysis applies. Two, determining the task transitions may not be as easy in assembly language as in FORTRAN [1,12]. In the parallel task recognizer the parallel

processable task graph requires that a determination be made when the output or result of one task is used by or input to another task. This is fairly easy in FORTRAN, because memory locations are stated specifically in the instructions [12]. It is not so obvious in assembly language, because many transitions depend on more subtle conditions such as flags set or interrupts enabled. Although these can be determined it is not always an easy matter of scanning. The recognizer will require more sophistication. But there are other considerations besides language.

Why use one time preload scheduling instead of dynamic scheduling as is done with some high level languages? It is because a microprocessor is usually driving a dedicated system, no matter how general its structure may be. It does not need the flexibility provided by a dynamic scheduler nor the associated overhead. By using one time scheduling, optimization cost is spread over thousands or millions of program executions [1]. According to the author's preliminary investigation with the NATIONAL SEMICONDUCTOR microprocessor family, the processor speed does not permit dynamic scheduling at the user program level that achieves any meaningful benefit if it is indeed possible. This conclusion is supported by others [8].

Dynamic scheduling with one processor acting as an interpretation or scheduling unit for other execution units and transfer units has been suggested by Lorin [10]. Because non-microprogramable processors are too slow to function as the scheduling unit, this function was relegated to the assembler for a one time schedule at load time. But even this method requires much analysis time. Therefore it is very

desirable to assure some hope of success. This is the reason for the suitability checker [1].

The suitability checker is important, because the parallel task recognizer requires large arrays with the number of elements equal to the square of the number of executable statements in the source program. This not only requires a large amount of memory, but requires more execution time to manipulate the arrays. The suitability checker, however, requires significantly smaller arrays. The largest is only four times the number of executable statements. Also there are two reasons why the parallel task recognizer alone is not sufficient for analyzing assembly language. First, it does not reveal anything about loop structure that would aid subsequent checking for loop iterations. Secondly, it requires more array manipulation for assembly language to find loops, because they are not explicit as in FORTRAN. But, all information required is available in assembler pass one, so it can make a determination before using the parallel task recognizer. The results could be indicated and decisions requested interactively or made based on a predetermined value.

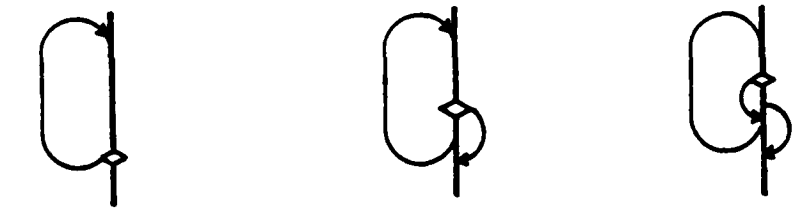
Assumptions for Assembly Language Suitability Checker

Two main assumptions were made to implement the suitability checker with respect to INTEL 8080 assembly language. These assumptions involve determining instruction types and recognizing loops. The INTEL 8080 was chosen because of experience with it, and because it is one of the most widely used microprocessors. These assumptions should apply to most any microprocessor language for recognizing instruction types and

recognizing loops. Recognizing instructions is obviously machine language dependent. The decision to use op-codes or mnemonics to recognize instructions depends on how op-codes are grouped and how well mnemonics relate to a class of instructions based on the variables in Ramamoorthy's suitability formula. The easiest method should be used for the machine in question. For the INTEL 8080, checking op-codes works very well to recognize instruction types. See Appendix D. But we must also be able to recognize loops.

Assume for assembly language that any backward jump is a loop. Although this is not the same as a DO loop, assume that these loops are for a known number of iterations. If necessary, some form of check can be done later to determine which loops are for indefinite iterations.

All types of FORTRAN-like loops can be described in assembly language with three basic constructs shown in Fig. 2. Three loop classes are necessary because loops are classified structurally by their entry and exit. For a simple loop the conditional jump provides entry and exit. For an intermediate or complex loop the unconditional jump back always provides a possible entry. The conditional jumps around the jump back provide a possible exit. If either entry or exit is not possible, there is probably a logical error. Why is it necessary to check for structure? First, because overlapped loops are not allowed in this analysis. Secondly, because analyzing structure makes some forms of error detection possible by determining if the minimum structure is not present. Thirdly, because it will allow later analysis of factors affecting iterations for scheduling considerations. After analyzing the



CLASS 1
SIMPLE

CLASS 2
INTERMEDIATE

CLASS 3
COMPLEX

CONDITIONAL JUMP

UNCONDITIONAL JUMP

All loops may be represented as one of these three classes. The class determination factors are the entry and exit to the loop. Other jumps may be present but are not required. If the minimum conditions are not satisfied, there is probably a logical error.

OVERLAPPED

Overlapped loops are not allowed, because they cannot be considered as either a single task or two discrete tasks by the suitability checker or the parallel task recognizer. They will be noted by the suitability checker. The overlapped loops would cause the results to be invalid, unless the overlapped loops are nested inside a third loop. In this case they would make no difference, because all tasks included in the outer loop would be considered a single task.

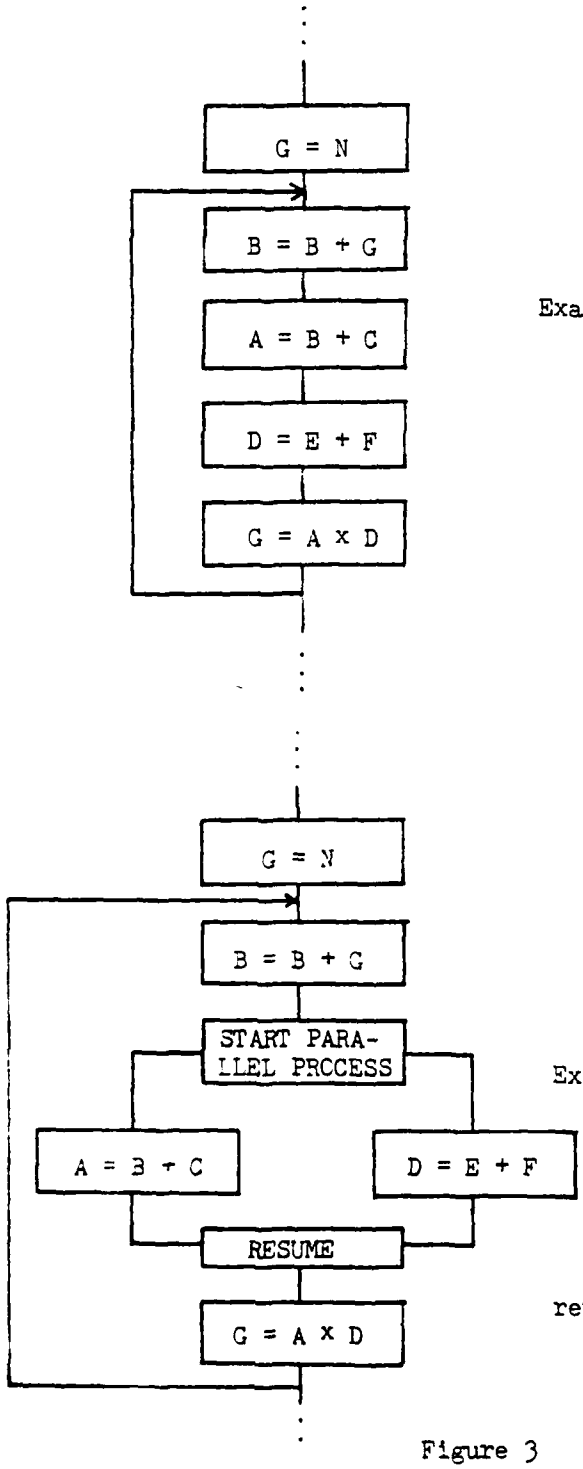
Figure 2

FORTRAN - Like Loops in Assembly Language

loops with respect to structure, what is the significance of other jumps?

There are three other situations wherein jumps are related to the loops: jumps out, jumps around, and internal jumps. Jumps out of the loop in excess of minimum requirements are associated with the loop. Possible transitions are noted, but the extra jumps are not significant to the structural classification. Jumps around the loop are not associated with the loop, because they eliminate it as a task, but the fact is noted and analyzed for possible errors. Other internal jumps in excess of minimum requirements are associated with the loop, because the recognizer will treat the whole loop as a single task. Also, each loop could be further analyzed as a subprogram to check possibility for parallel processing within the loop. See Fig. 3.

Based on these assumptions it is possible to define a loop, what loop classes are present as shown in Fig. 2, and how other jumps relate to the loops. Basic rules are established for checking type and number of instructions, recognizing loops, and deciding how they affect the subject program in terms of its suitability for parallel processing. The next chapter discusses the suitability checker algorithm.



Example of a loop as a single task to find G.

Example of using two processes to find A and B simultaneously, then find G after returning to a single process.

Figure 3

Parallel Processing Within a Loop

III. ALGORITHM FOR SUITABILITY CHECKER

The assembler suitability checker is based on Ramamoorthy's research discussed in the previous chapter. The suitability checker presented herein has been adapted for use with assembly language. These changes to the cross assembler are shown in Fig. 4. During assembler pass one it scans the instruction's op-code to count different types of instructions for the suitability factor, SF. If a jump is found, it checks to determine the type of jump and build a jump table. At the end of pass one, the loop analysis process requires scanning from the end of the source program to the beginning to see if minimal requirements for loops are met, check for possible errors, calculate the SF, and find the nodes of the loop free program graph (LFPG). The output is shown in the short example listing in Fig. 5.

The loop analysis is the most lengthy and complex portion of the algorithm. The source code is checked backwards, because it is possible to work back from the jump and determine what is included in the loop. Multiple jumps cannot start at the same point, but they can terminate at the same point. The objective is to associate as many jumps as possible with loops. Others not associated must be isolated and are, therefore, negative factors in the suitability equation. This is so, because every statement contained in a loop is treated as a single task by Ramamoorthy's parallel task recognizer.

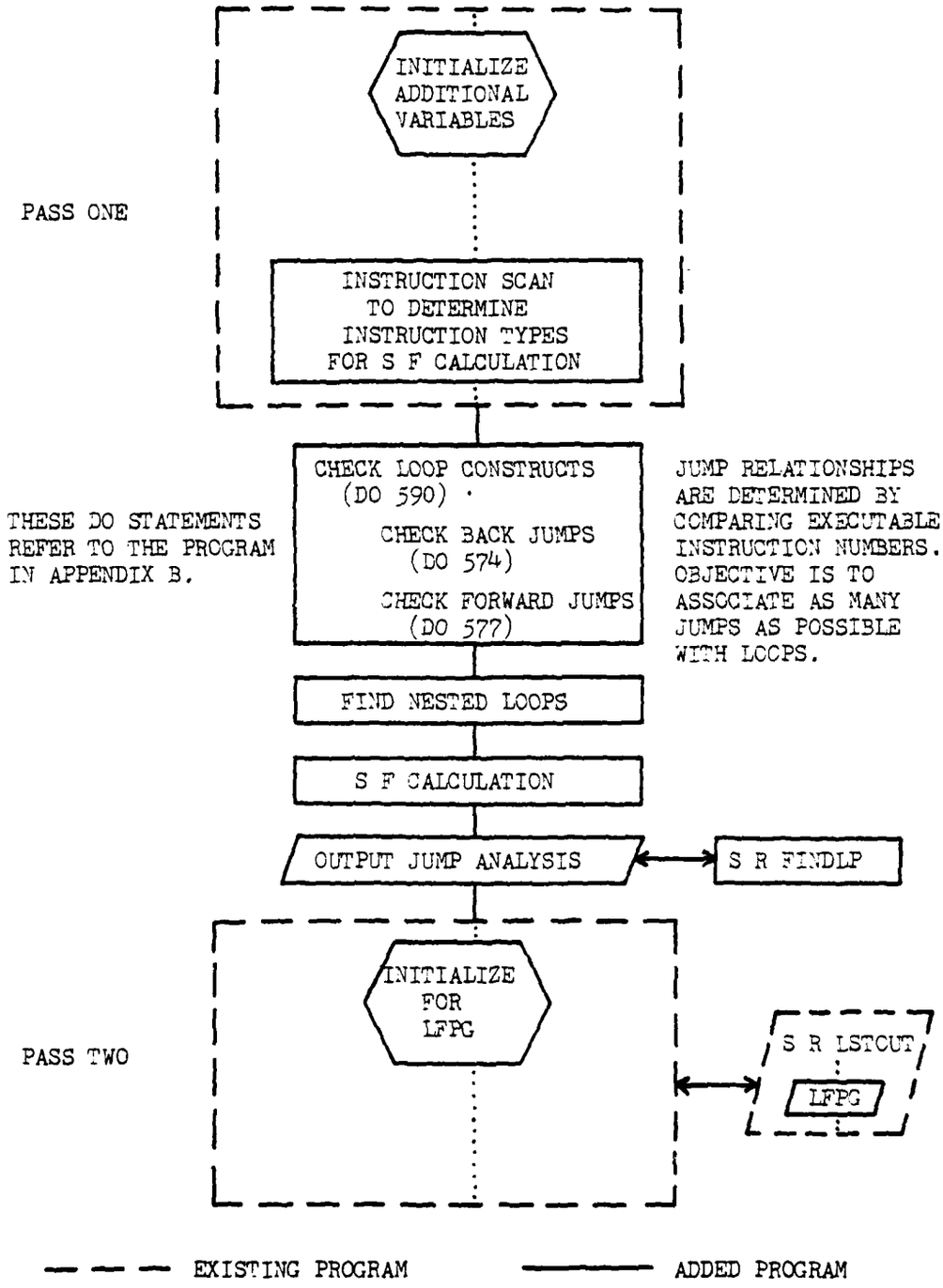


Figure 4

Modifications to the Assembler Program

Figure 5

Analysis for a Simple Microprocessor Program

This chapter explains how the suitability checker is incorporated into the assembler. Although it is possible to quickly find jumps and other types of instructions to count for the SF calculation, more analysis is required to determine nested loops, jumps not associated with loops, and number of instructions within loops. This is the purpose of the loop analysis. After finding the types of instructions, all of the loops must be checked to determine their construction. This is a means of determining the relationships of all jumps according to the three loop constructs defined in the previous chapter. Each jump is first checked to see if it is already associated. If not, it is checked to find if it is forward or back. If back (a loop), all forward jumps are checked against it to see if any forward jumps go around the loop, jump out of the loop, or jump into the loop. This information is used to determine the loop class or any errors. On the other hand, if the jump being checked is forward, checks of all forward jumps and each subsequent loop are made to see if it jumped around any loops, if it jumped around any other forward jumps, or if any forward jumps jumped around it. In this way, the relationship of all jumps is determined and analyzed for errors. Then the identifications for nested loops, unassociated forward jumps, and instructions within loops can be made.

The remainder of this chapter discusses in detail the algorithm necessary to modify the main assembler program, modify the source list, and add one subroutine to find loop numbers. This includes discussion of common areas and variables, instruction scan, loop structure check, nested loop check, SF calculation, jump analysis, showing the nodes of

LFPG, and subroutine FINDLP. See Appendix B for the assembler modification instruction listing.

Common Areas and Variables

Two named commons, LOOP and INST, were added to the main program. The array sizes chosen have proved workable, but could be increased if desired. Common LOOP contains information about loops found in the source program. It includes four variables. LPMAX is the total number of loops. Array LP (100) is the loop table which contains the executable instruction number of each loop's jump instruction. LPNO, loop number, is the pointer to the loop array, LP (100). LODES is a pointer to the loop table entry whose destination is currently the lowest during the nest check. The other common, INST, contains information about instructions. It has five variables. The array JPX (1000,2), for jump/execution number, holds the jump number and executable instruction number for each line of source code. The array JP (200,4), for jump table, holds four data for each jump. That is: 1) the executable instruction number of the jump, 2) the executable instruction number of the destination, 3) the type of jump or class of loop, and 4) the pointer to the associated loop. MSTOPN, for stop number, and MSTOPD, for stop dot, are used in printing the source list to show the destination and jump for each outside loop. LFPG is the node number of the loop free program graph. The array JPDES (200), for jump destination, is used only by the main program as a table of jump destinations. It is not in a common. It holds the symbol table pointer for each jump's destination. This is required, because the executable

instruction number of the forward jump's destination is not known until the end of pass one. For a list of all variables, see Fig. 6.

Counters and Other Variables

The suitability factor, SF, requires seven counters:

$$SF = ((NIO+NAL+NC+NB-NF)/(NX-NL)) - (NL/NX)$$

NIO is the number of input or output instructions (equivalent to FORTRAN reads and writes). NAL is the number of arithmetic and logical instructions. NC is the number of unconditional calls. NB is the number of loops (backward jumps). NF is the number of forward jumps not associated with a loop plus the number of conditional calls. NX is the total number of executable instructions. NL is the total number of instructions contained in outside loops. Note that this includes all instructions in loops, but instructions in nested loops are not counted more than once. By comparing these seven variables to Ramamoorthy's formula on page 7, the relationships may be noted. Six additional variables are required to modify the program.

NJ is the jump table pointer to array JP (200,4). NJMAX is the total number of jumps. KPTR is an additional symbol table pointer that can be used to determine a label location without disturbing the original symbol table pointer, SMBPTR. K, the instruction type, is part of the original assembler. It is used with the original ICODE, the instruction op-code, to determine the variables for SF. See Appendix D for explanation of op-code groupings. KDEST is used as the address of a jump destination, so a label location can be noted without disturbing the original symbol table pointer. The next section explains how all these variables are used in the analysis.

VARIABLE	DESCRIPTION	INITIAL VALUE	USED BY
COMMON /LOOP/			
LPMAX	TOTAL NUMBER OF LOOPS	0	PASS 1
LP(100)	LOOP TABLE	0's	PASS 1
LPNO	LOOP TABLE POINTER	0	S R FINDLP
LODES	LOOP TABLE POINTER TO LOW DESTINATION	LPMAX	PASS 1
COMMON /INST/			
JPX(1000,2)	JUMP NO. & EXEC. NO. FOR EACH STATEMENT	0's	PASS 1 S R LSTOUT
JP(200,4)	JUMP TABLE	0's	PASS 1
MSTOPN	FLAG TO STOP LFPG NUMBERS	NX	S R LSTOUT
MSTOPD	FLAG TO STOP DOTS (LFPG NON NODES)	EX. NO. OF OUTSIDE JUMP	S R LSTOUT
LFPG	LFPG NODE NUMBERS	0	S R LSTOUT
JPDES(200)	DESTINATION TABLE (SYMBOL POINTERS)	0's	PASS 1
NJ	JUMP TABLE POINTER	1	PASS 1
NJMAX	TOTAL NUMBER OF JUMPS	NJ-1	PASS 1
KPTR	AUXILLIARY SYMBOL TABLE POINTER	CURRENT S. T. POINTER	PASS 1
SMBPTR	ORIGINAL SYMBOL TABLE POINTER	0	PASS 1
K	ORIGINAL INTEL INSTRUCTION TYPE	0	PASS 1
ICODE	ORIGINAL OP-CODE	TABLE	PASS 1
KDEST	AUXILLIARY DESTINATION ADDRESS	CURRENT DESTINATION	PASS 1

Figure 6

Variable Initialization

Instruction Scan

The original assembler program is altered so that the scanner (program label 500) checks each source line after the assembler finishes and before the next source line is read. See Fig. 7. The scanner checks the op-code, ICODE, which is already available to determine the instruction type for SF calculation. For the INTEL 8080 language it was found convenient to check op-codes because of the way they are grouped. See Appendix D. The type of instruction can be determined by checking within a range of op-codes. If it is a jump, note jump type as follows:

1. Type 1 is conditional backward;
2. Type 2 is unconditional backward;
3. Type 6 is conditional forward;
4. Type 8 is unconditional forward.

Forward jumps are distinguished from backward jumps by the fact that only backward jumps have a destination defined prior to their encounter by the scanner. This is determined by checking the symbol table. If the destination is undefined, either the jump is forward or there is an error. Based on this, construct the initial jump table in array JP. This includes the jump's execution number, destination if backward or KDEST if forward, and jump type. Backward jumps only (loops) are associated with themselves. Continue by counting executable instructions, but comments and pseudo op's are not counted. Then return to the assembler to read the next source line at program label 1. At the end of pass one, signified by the END pseudo op, find the executable instruction number of the destination for all forward jumps. See Fig.

FOUR ENTRY POINTS TO INSTRUCTION CHECK FROM THE ASSEMBLER, WHERE IT FOUND EXECUTABLE INSTRUCTIONS. → 500

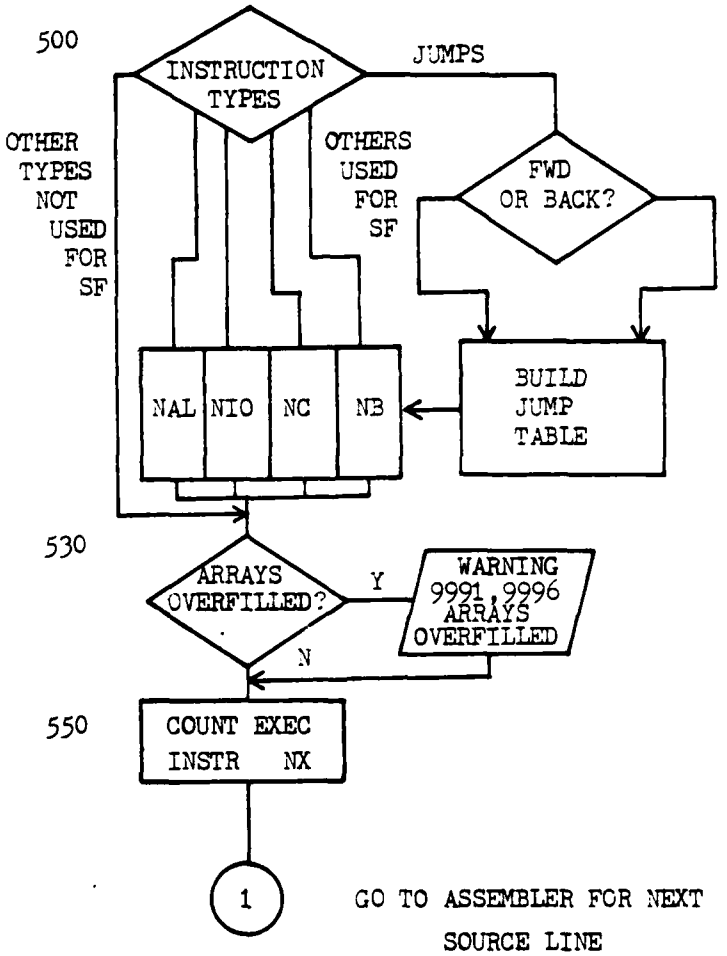


Figure 7

Flowchart for Instruction Scanner

8. Prior to pass two, check loop constructs, find nested loops, and do the jump analysis.

Checking Loop Constructs

This discussion is supplemented with flowcharts throughout the following pages. Also the reader may wish to refer to Appendix B. All program labels and format statements refer to the program listing of assembler modifications in Appendix B.

Loop constructs must be checked to classify loops and find any structural errors as well as to associate all possible jumps. This process is called DO 590 in the program. See Fig. 9. The index is MC. Note that since it is desirable to check backwards, the index is manipulated to do this and the reverse index is MJ. First, in the jump table, check if the jump is already associated. If it is, go to 590, because that means it has already been checked. If not, branch depending on whether the jump is forward or back.

If the jump checked by DO 590 is backward signifying a loop, go to label 562, the start of DO loop 574. This is shown in Fig. 9. All forward jumps are checked against the loop for possible association with it. To be associated, a jump must start in the loop. If it does start in the loop, there are two possibilities. If the jump stays in the loop, it is not significant to the structural classification unless it jumps around another unconditional jump out. If a jump does exit the loop, it is associated, and the loop is checked for entry and exit. This is done by checking all previous jumps out to make sure one is not unconditional. This could preclude entry to the loop. If such an

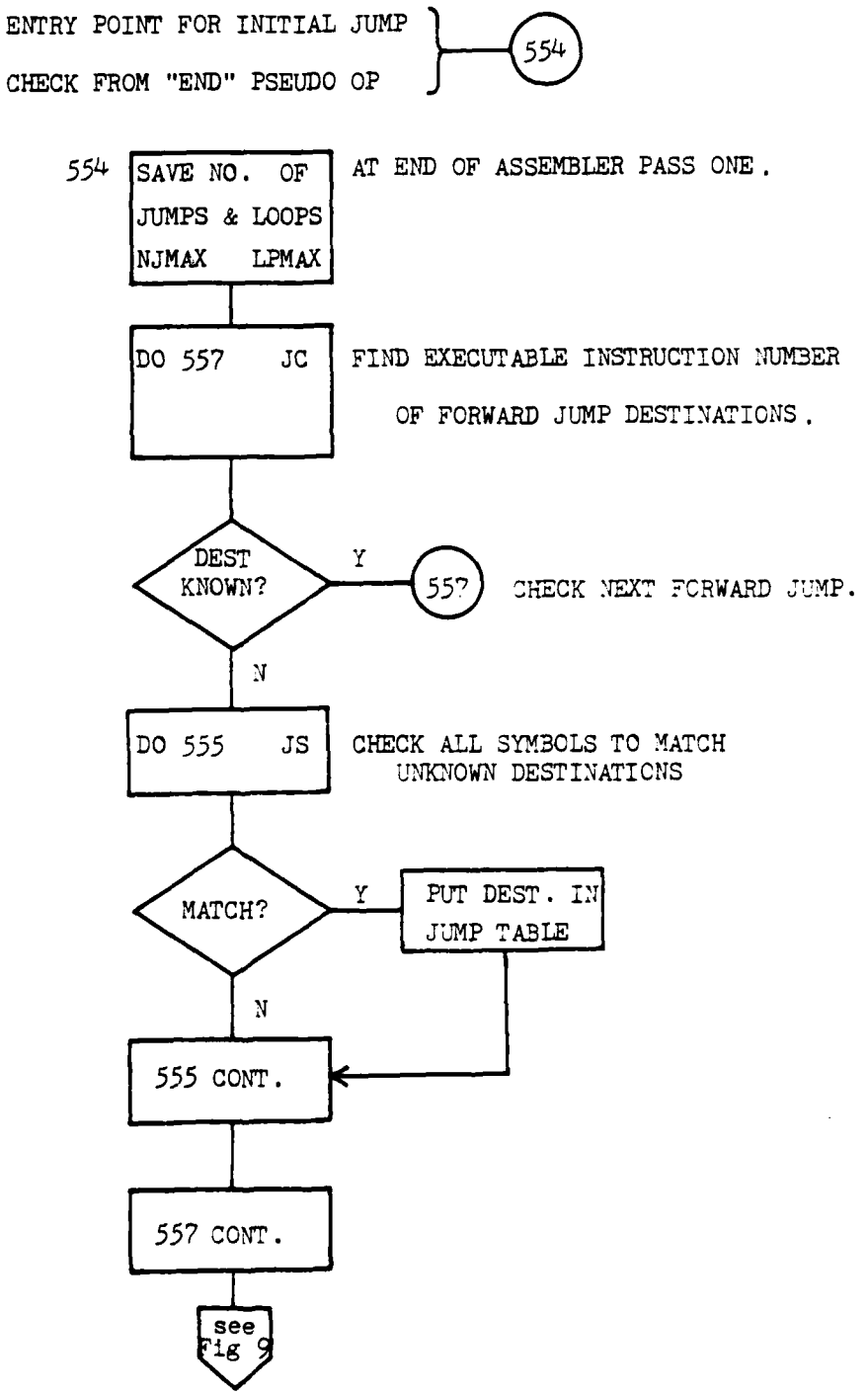


Figure 8

Flowchart for Finding Forward Jump Destinations

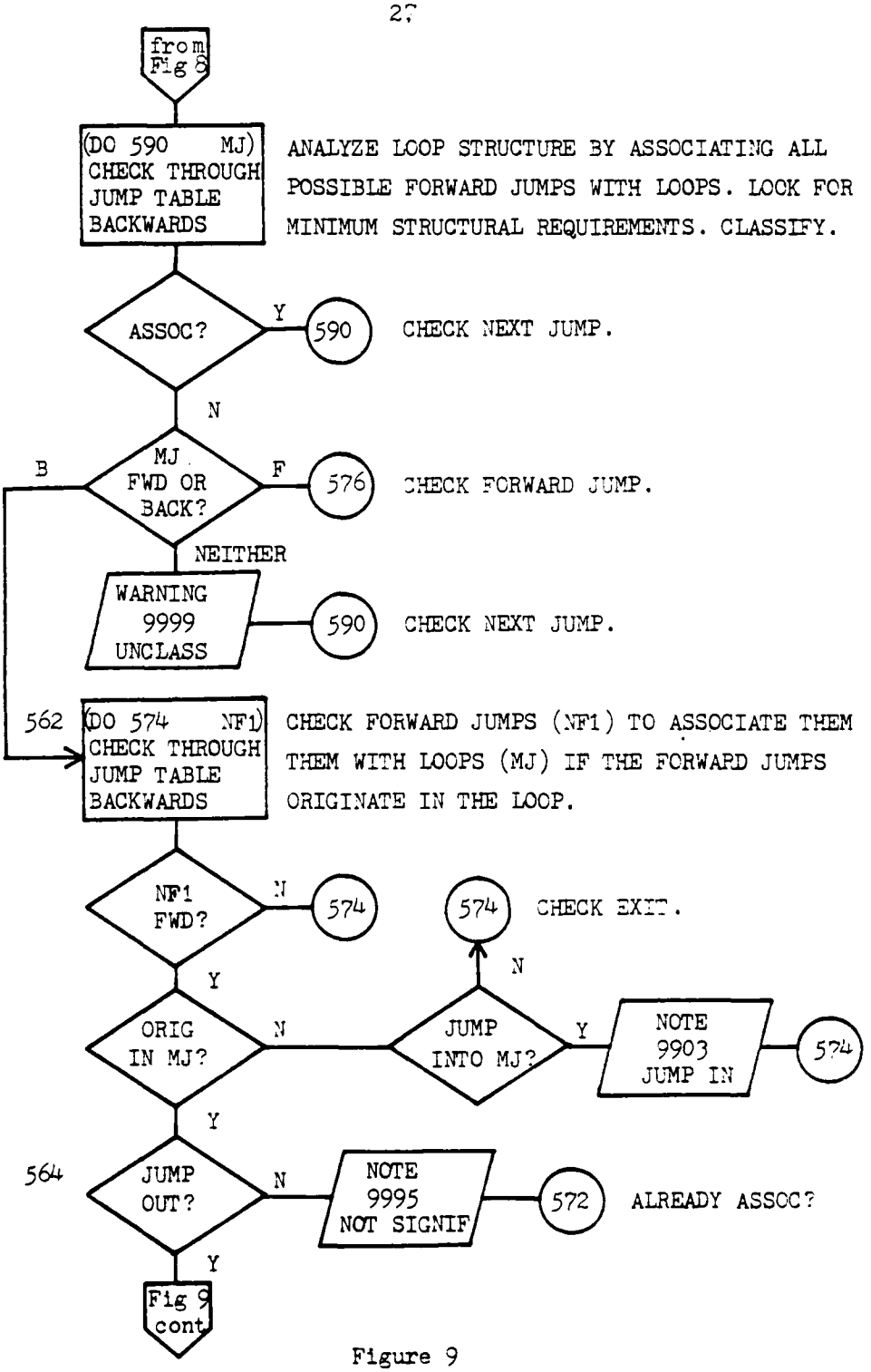


Figure 9

Flowchart for Checking Loop Constructs

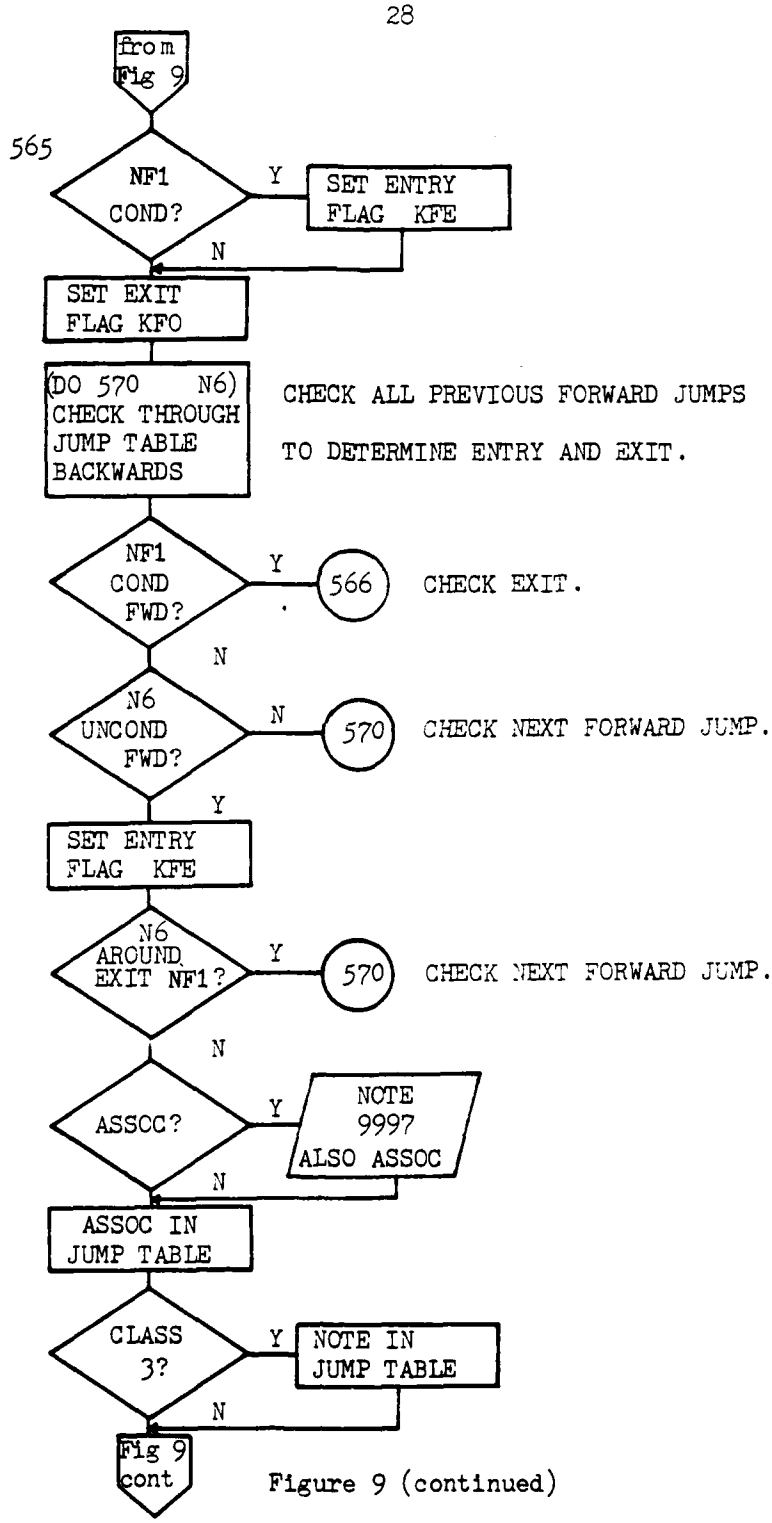


Figure 9 (continued)

Flowchart for Checking Loop Constructs

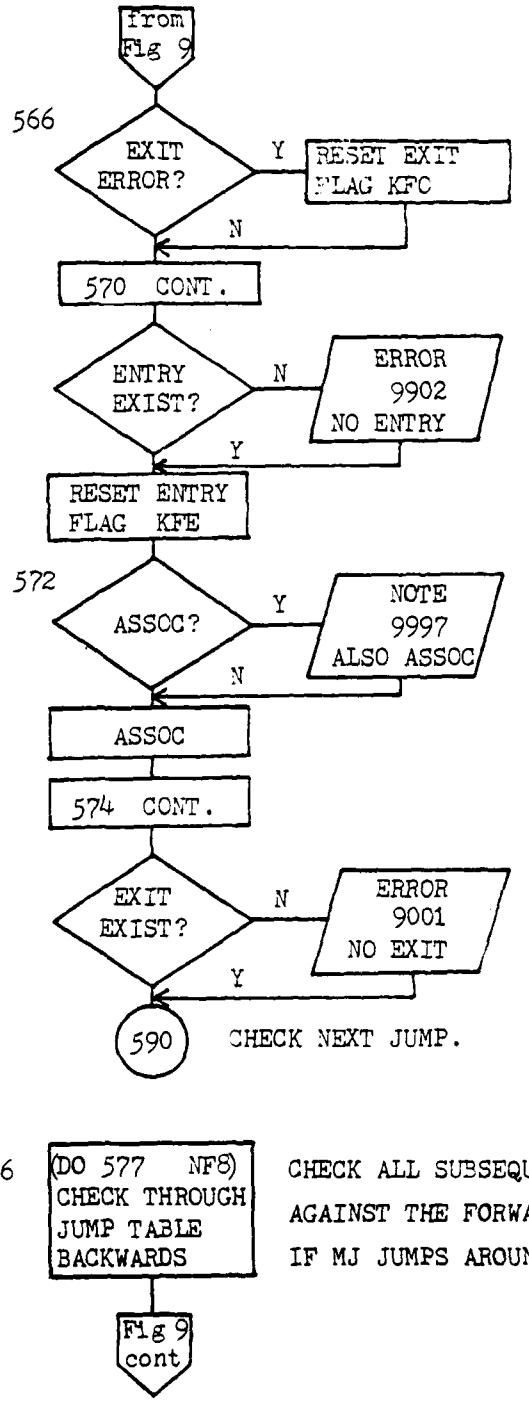


Figure 9 (continued)

Flowchart for Checking Loop Constructs

unconditional jump out is found, a further search is made for a conditional forward jump around it. This complex construct identifies a Class 3 loop as shown on the second page of Fig. 9 and in Fig. 2.

Each exit is also checked to see if it is already associated. If it is, this implies the presence of nested or overlapped loops, because one jump exits at least two loops from the same point. Multiple associations are noted, because the jump table permits only one association to be stored for each jump. The loop class is also stored in the jump table. Entry and exit errors, if any, are noted before checking the next jump in DO loop 590. This process of checking loops is shown in the first three pages of Fig. 9.

The last two pages of Fig. 9 show the process for checking forward jumps identified by DO loop 590. For this case, all subsequent loops and previous forward jumps must be investigated to determine relationships with the forward jump. If the forward jump goes around a loop, a negative association is made with the loop to show this in the output. Such a forward jump is still counted as unassociated in the SF, if it is not part of any other loop. If the forward jump goes around another jump, there are two possibilities. If the jump around is conditional, it is simply noted. But if it is unconditional, it is noted as a possible error, because it could preclude the other jump from being executed. This is a useless situation and a logical error. If, at the end of DO loop 577, the forward jump is not associated at all, it is associated with itself to show that it is an isolated forward jump. This is the end of DO loop 590, the loop construct check. Using the

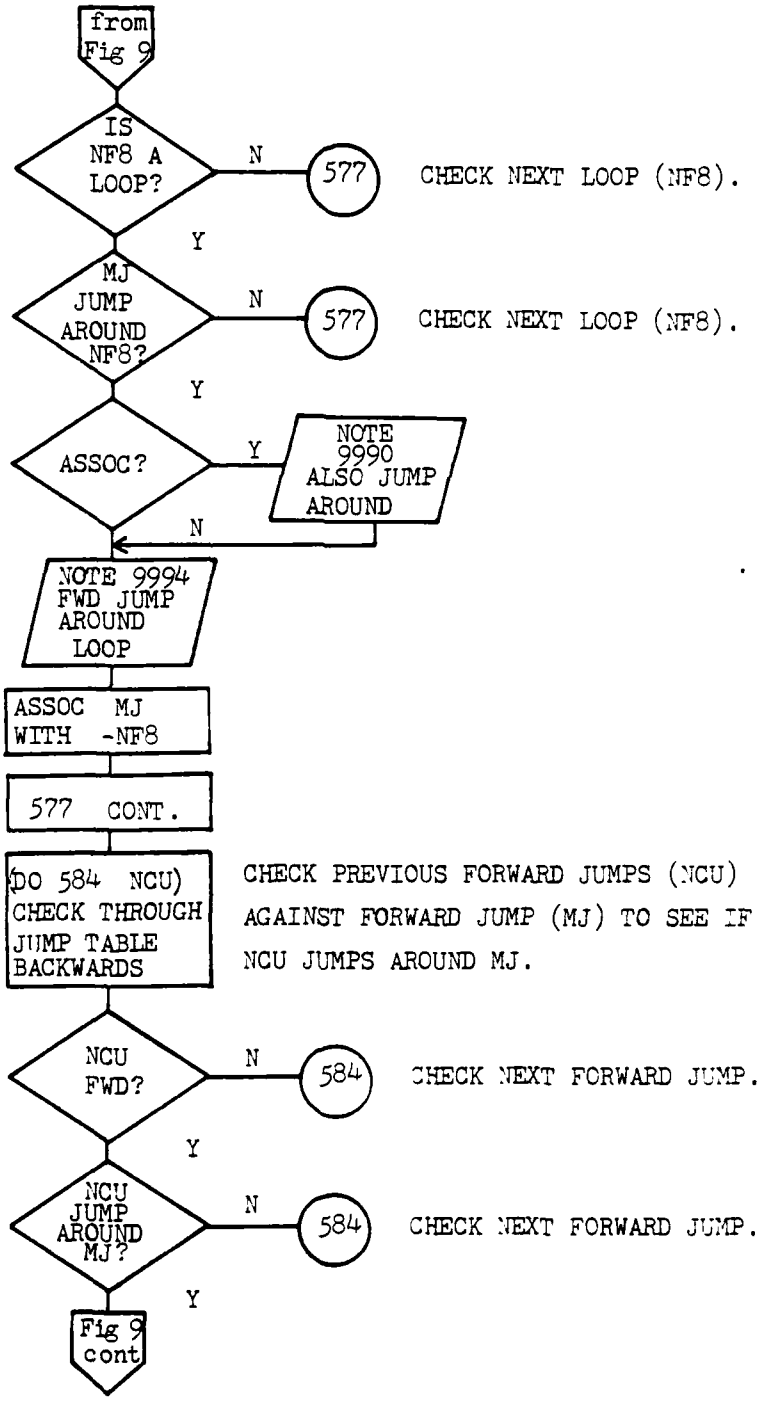


Figure 9 (continued)

Flowchart for Checking Loop Constructs

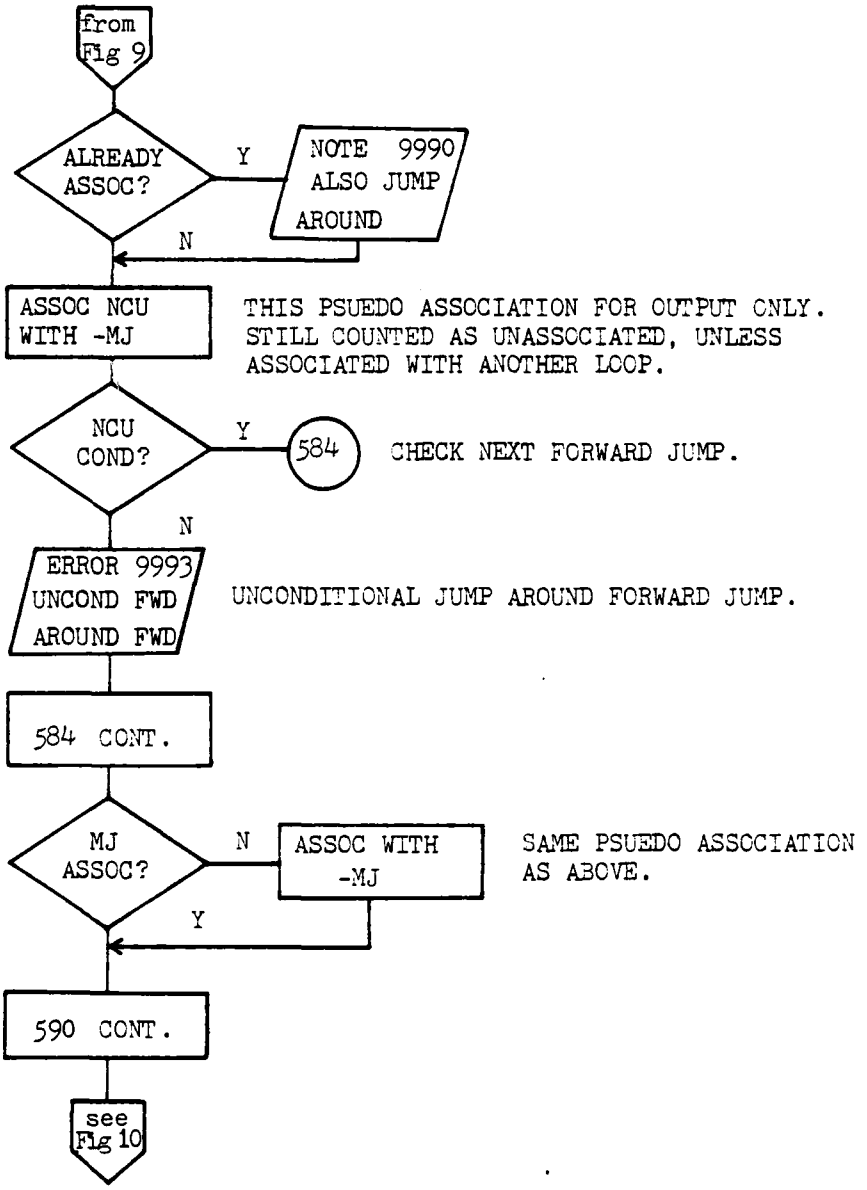


Figure 9 (continued)

Flowchart for Checking Loop Constructs

information gained here, it is now possible to make a positive check for nested loops.

Nest Check

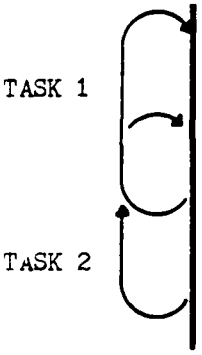
The purpose of the nest check is to find nested loops and overlapped loops. See Fig. 10. This is done by using the loop table as a pointer to check through the jump table backwards. Each loop is checked against the previous loop and the outermost loop. The objective is to see if the range of executable instruction numbers for the previous loop is a subset (nested) of the others or an intersecting set (overlapped) of the others. This checking process is done in a loop called DO 588 as shown in Fig. 11. KC is the loop index which points to the loop currently being checked. LODES points to the loop having the lowest numerical destination of those already checked (the outer loop). To start the check, the last loop is designated the outer one, because there can be no subsequent loop to contain it. During the check there are four possible situations. The previous loop may not be nested. The previous loop may be overlapped with the current loop. The previous loop may be nested in the current loop. Or, the previous loop may be nested in the outer loop, LODES. For each of these situations, a message is shown except for the case when a loop is not nested. If the nests were checked completely to the inner loop for each iteration of DO loop 588, the process could become very complex. To avoid this, the current loop, KC, is checked only against the previous loop and the outermost one. This requires checking each current loop to see if it is the new LODES, but this is much simpler than trying to track a nest down

SEQUENTIAL LOOPS:

Task two comes after task one. Tasks are distinct. They can be shown as separate nodes on the loop free program graph.

OVERLAPPED LOOPS:

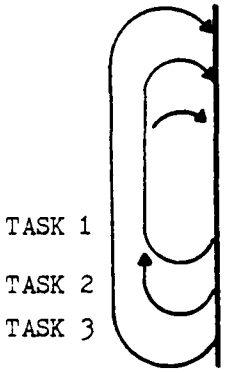
NOT NESTED



CANNOT BE ANALYZED

Tasks one and two are not distinct but converging. They cannot be shown as two separate nodes on the LFPG.

NESTED



CAN BE ANALYZED (ALL TASKS CONSIDERED PART OF TASK THREE).

NESTED LOOPS:

Tasks one and two are converged. Together they are shown as a single node on the LFPG.

Figure 10

Task Convergence and Overlapped Loops

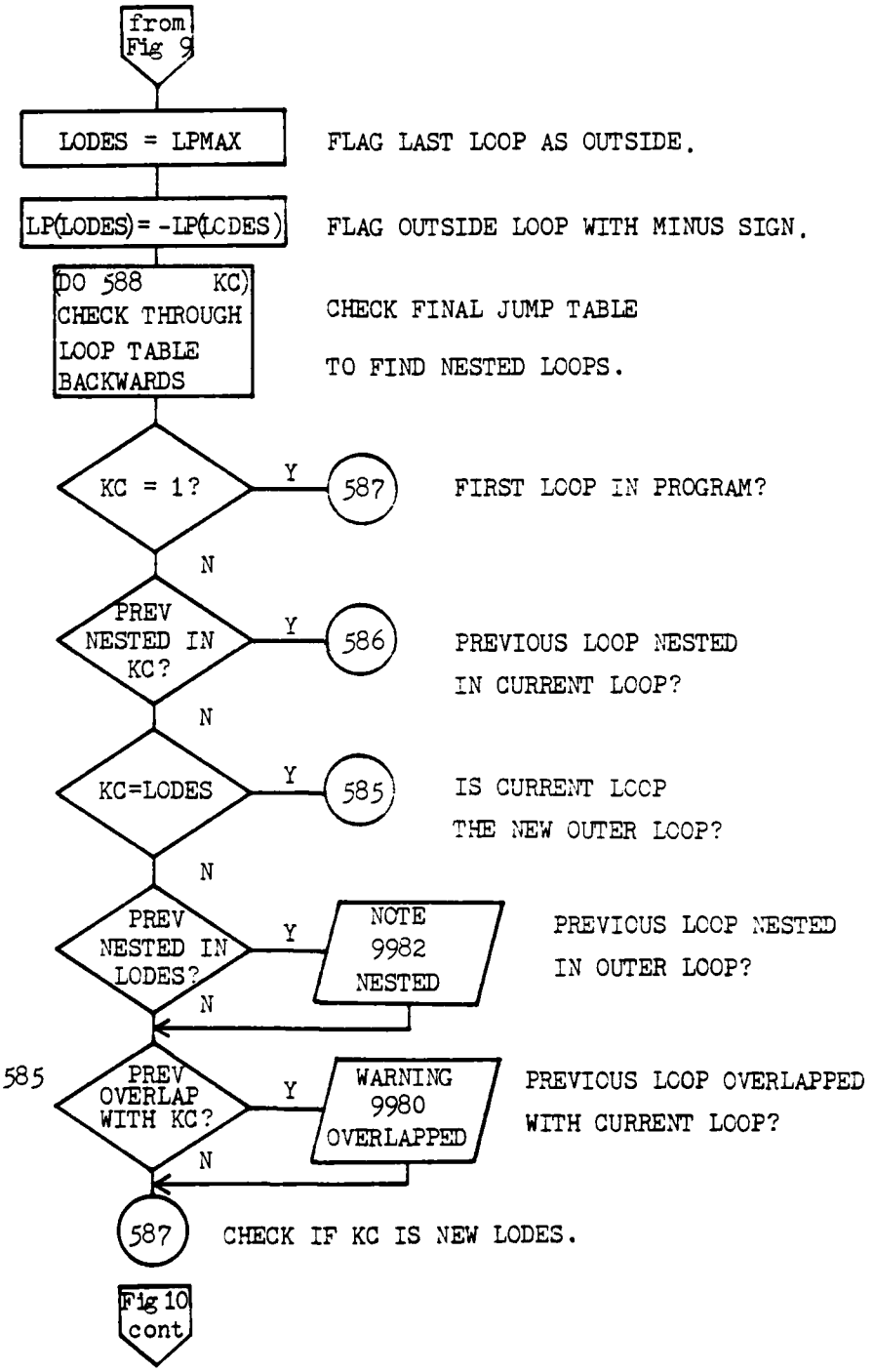


Figure 11

Flowchart for Nest Check

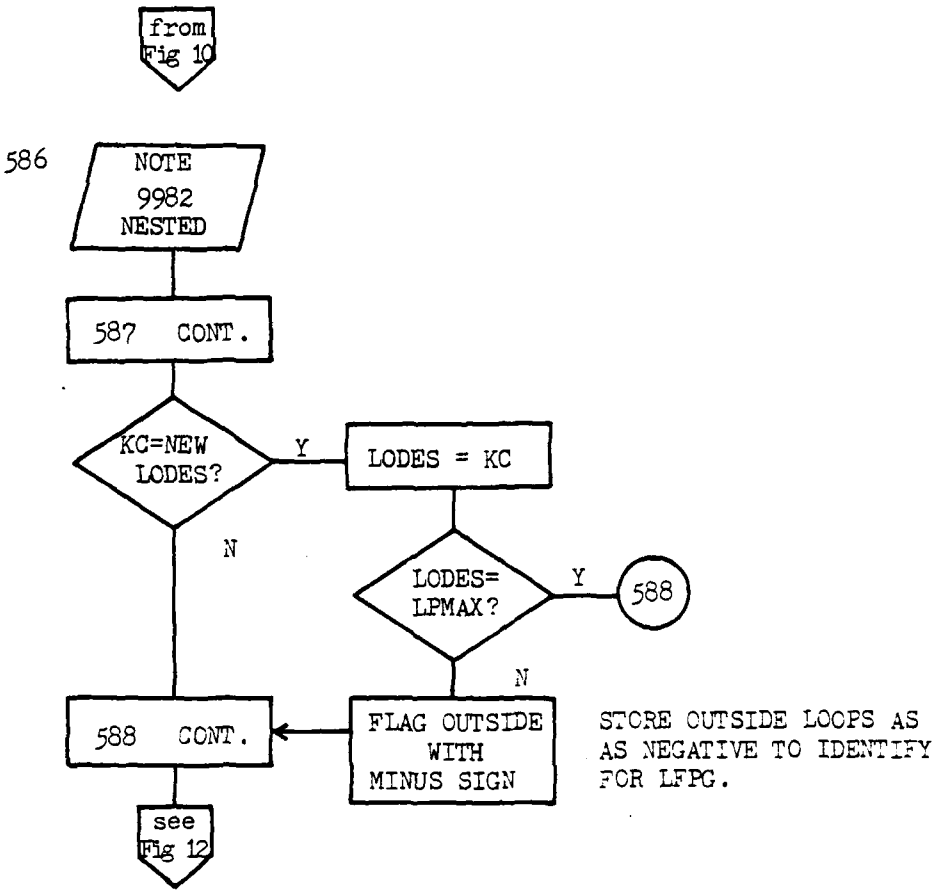


Figure 11 (continued)

Flowchart for Nest Check

to its innermost loop in a single iteration of DO loop 588. Outside loops, regardless of whether they contain nested loops, are flagged for later use with the LFPG. This concludes the discussion of the nest check.

Final Checks

Three final checks are necessary to calculate SF. Check the final jump table first for forward jumps with no positive association. Increment NF for each of these isolated forward jumps. Next find NL, the total number of instructions in loops, by subtracting the executable instruction number of the destination from that of the jump. This result for each outside loop is added to find NL. This is not done for nested loops. Otherwise, NL could be larger than NX, the total number of instructions. This would cause SF to be negative. Also, NB must be discounted for each nested loop to show the number of outside loops only. Prior to the SF calculation, check if NX = NL. In this case, show the unsuitability for parallel processing (format 9010). Otherwise, SF is calculated and output with the jump table. These three checks are shown in Fig. 12.

Jump Analysis

The jump analysis checks only forward jumps to find those unconditional ones which circumvent other instructions. This is done by trying to find another forward jump to the next instruction. If such a path cannot be found, a search is made for a loop back to that next instruction after the unconditional forward jump. If no path is found, a possible error message (format 9493) is shown. Note that in programs

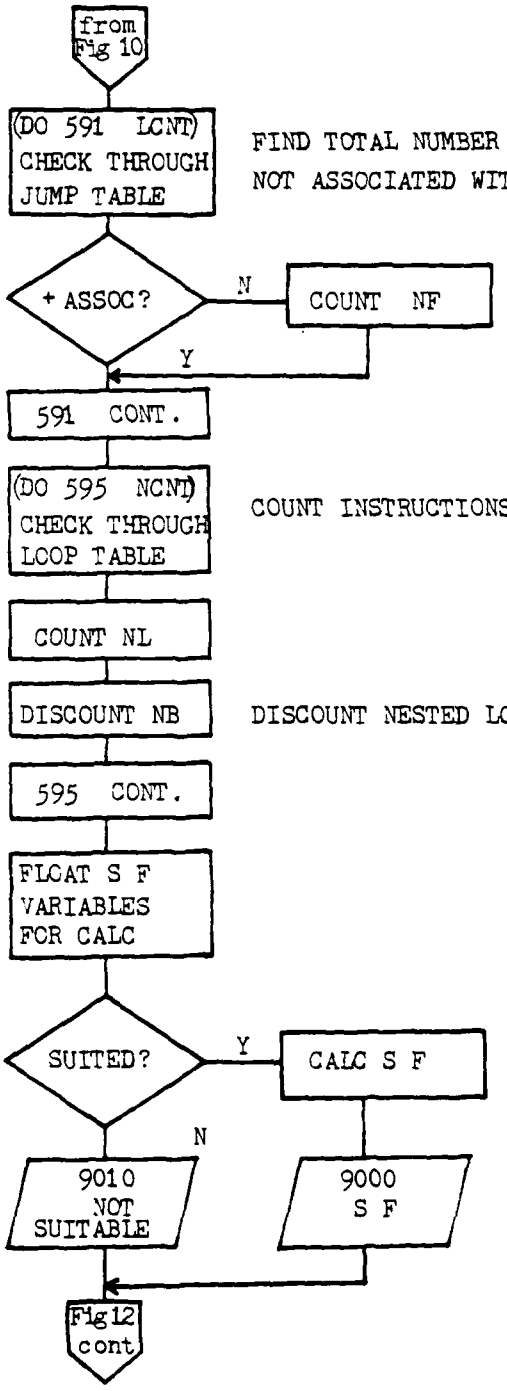


Figure 12

Flowchart for Final Checks and Jump Analysis

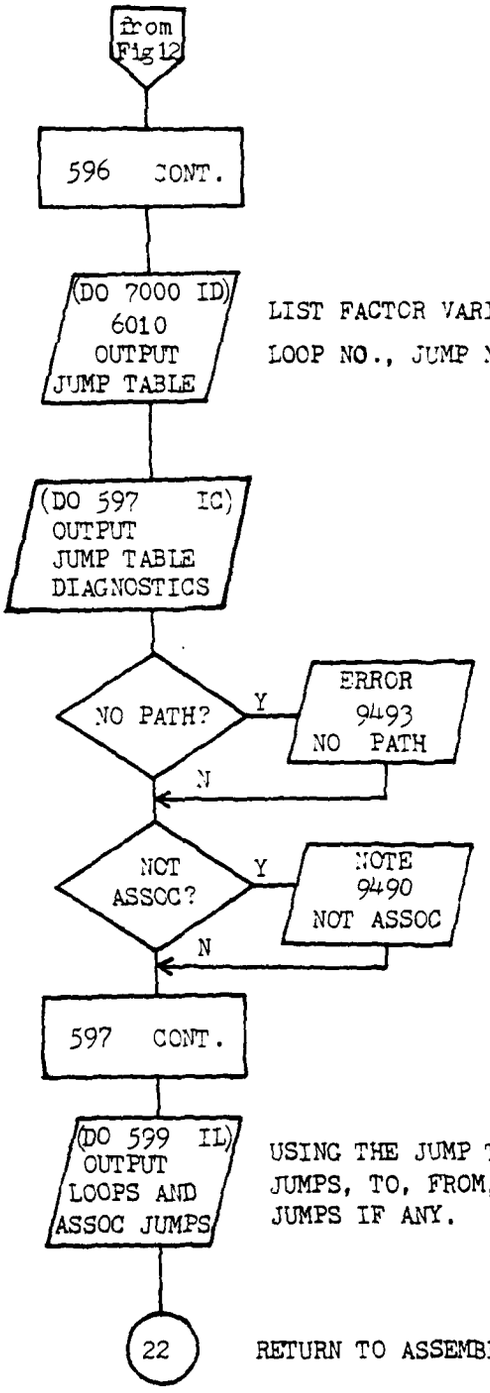


Figure 12 (continued)

Flowchart for Final Checks and Jump Analysis

which have vectoring (jump to an address that jumps to another address or has its own return mechanism), this will cause erroneous error messages, because the return is obscure. At label 5964 note jumps not associated with any loop. At label 597, the end of the jump analysis, write the headers for listing the loops. Then find and list each loop with its associated forward jumps, if any. If there are none, it will so indicate. This is the end of the jump analysis as shown in Fig. 12 on the previous page.

Loop Free Program Graph

The key to the Loop Free Program Graph (LFPG) is the loop table which has the outer loops tagged as negative by the nest check. This is used in the present version to print a notation on the source list showing the node numbers of the LFPG and non nodes as dots in a column between the jump number and the executable instruction number. This was shown in Fig. 5. This is done by subroutine LSTOUT at the same time it is making the source list. This capability is initialized by the main assembler program by setting MSTOPN to the number of the last loop in the jump table. MSTOPN is the flag that suppresses the LFPG node numbers within outside loops on the listing. When subroutine LSTOUT is called, the assembler pass two has been modified to determine if there are any loops. If there are none, it performs normally. If there are loops it sets a flag, MSTOPN. This will stop printing numbers and start printing dots at the beginning (destination label) of the outer loop. It also sets MSTOPD to stop printing dots at the end (jump) of that loop. See Fig. 13 and Appendix B.

EXISTING SUBROUTINE LSTOUT WRITES OBJECT CODE FOR LOADER AND SOURCE CODE FOR ASSEMBLY LISTING. IT IS CALLED FROM PASS TWO FOR EACH SOURCE PROGRAM LINE.

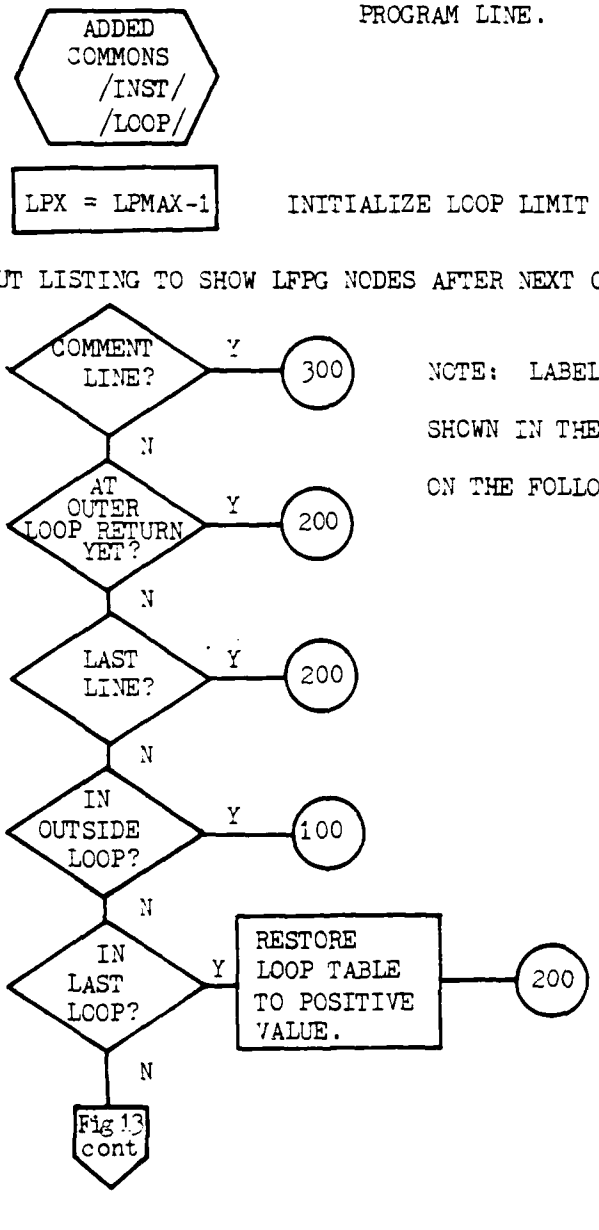


Figure 13

Flowchart for Subroutine LSTOUT Modification

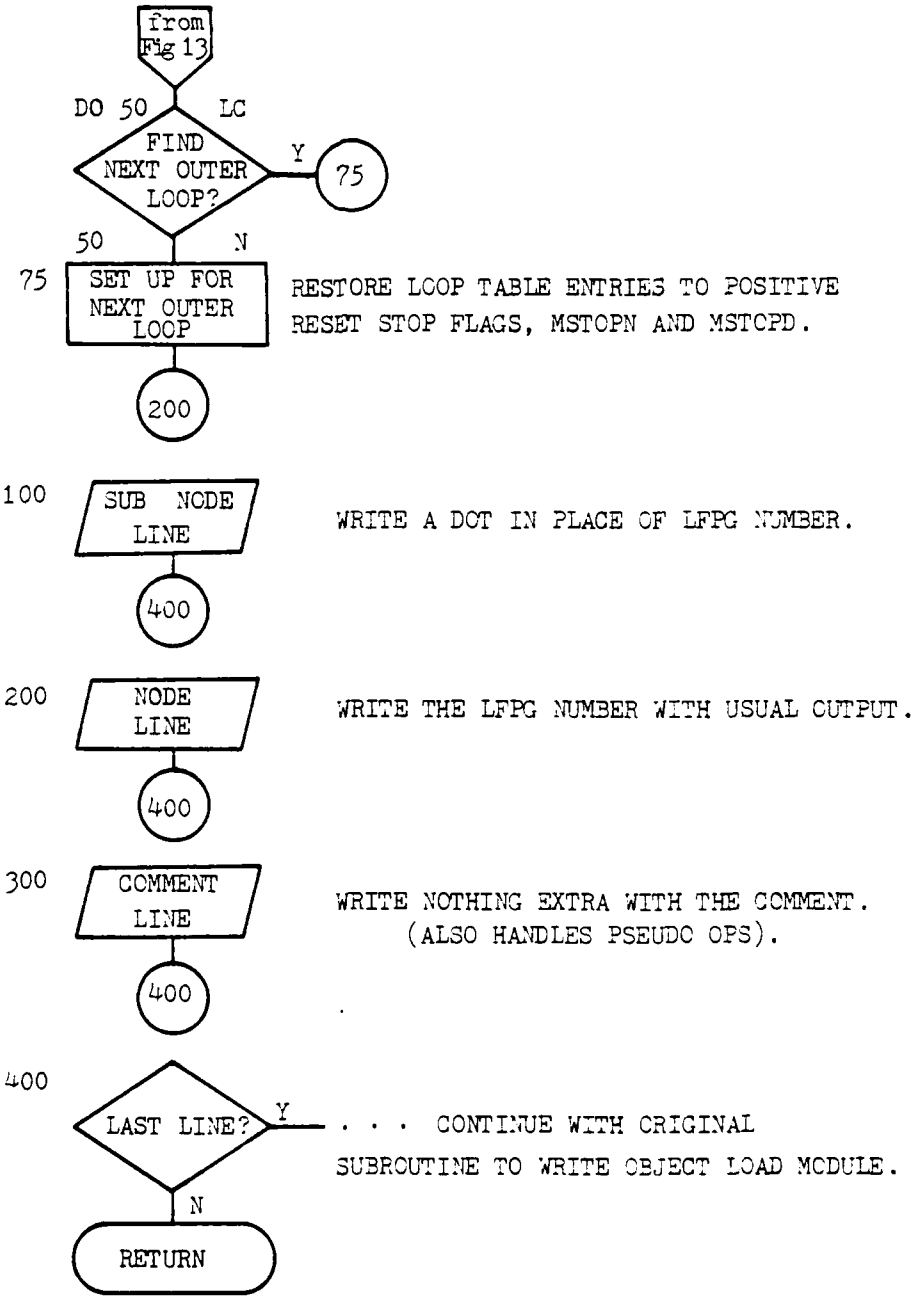


Figure 13 (continued)

Flowchart for Subroutine LSTOUT Modification

The subroutine LSTOUT has been modified by adding the COMMONS INST and LOOP. A variable LPX is initialized to point at the last loop, because it limits the loop index used to find the next outside loop. It checks while making the source list for comment lines or pseudo op's. Nothing extra is printed on these lines, because they have no significance for the LFPG. The node number is printed on the line if the executable instruction number is less than MSTOPN. If not and the line is part of a loop, a dot is printed. Else it is the backward jump, for an outer loop so the node number will be printed, and the flags are reset. This continues until the last executable instruction which must always be a node by definition.

The subroutine FINDLP has been added to find loop numbers from the loop table when given a jump number passed in the call. See Fig. 14 and Appendix B. This is necessary to provide comprehensive diagnostics that reference both the jump and the loop numbers.

This completes the discussion of the algorithms implementing the suitability checker, finding nested loops, and printing the LFPG. The next chapter discusses the interpretation of the suitability determination and the diagnostics.

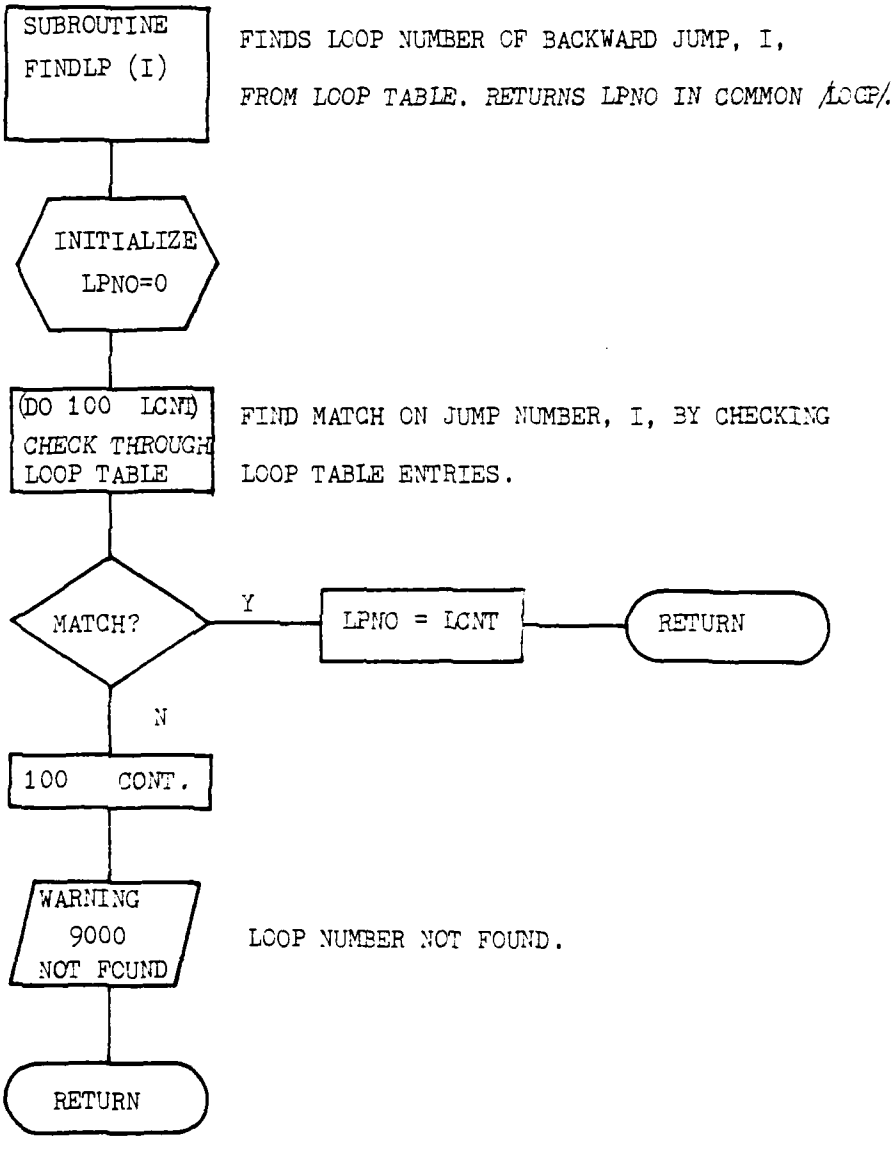


Figure 14

Flowchart for Subroutine FINDLP

IV. SUITABILITY FACTOR AND DIAGNOSTIC MESSAGES

This chapter discusses interpretation of the suitability factor, SF, and the error messages, warnings, and notes which are a useful by-product of the analysis.

Suitability Factor

Although SF is partly an empirical factor, it can be theortically justified as discussed by Ramamoorthy [1]. He discussed the value of SF = 0.3 as being a useful cutoff point. According to his observation and reasoning, a value greater than 0.3 indicates that a program has characteristics favorable for parallel processability. Therefore, the same value has been used as noted in the output. Further research may indicate a different value for the assembly language suitability checker because of two differences between this application and Ramamoorthy's. These differences concern loops.

Since Ramamoorthy was interested in dynamic scheduling, he included backward jumps of unknown iteration as a negative contributing factor. With assembly language, the predetermination of loop iterations cannot be made at this stage of analysis. Therefore, the assumption was made that all the loops are designed for a predetermined number of iterations. For one time scheduling of the program in the multi-microprocessor system, each partition will be scheduled to run on a processor until it is finished. For this reason, the schedule will be

based on earliest and latest possible task initiation times only; not dynamically based on how long the partition will take to execute. Of course, the execution time must be reasonably balanced with other partitions, but this is the indication given by the suitability factor. This is reasonable for nondynamic scheduling, because the final solution will eventually require some balancing or fine tuning. This would involve consideration of loops of indefinite iterations such as loop until interrupt. Based on the overall system, a decision would be made whether to allocate such a loop as a discrete task or part of a larger set of tasks. With respect to nested loops, this is a more complex problem.

Since Ramamoorthy's analysis does not allow nested loops and does not recognize loops of unknown duration, it really does not consider loop length except for the caveat that parallel paths must not be too long [11]. This is checked by referencing the number of instructions in loops. Nesting effectively increases the loop length as shown in Fig. 15. Because optimal length is relative to the length of other partitions, it is not possible at this stage of analysis to judge this. The number of instructions in the loop is only a rough indication. A short loop executed many times could run longer than a long loop executed only a few times. Therefore, it is necessary to examine not only loop length, but also iterations for the outer loops and any inner loops. It is not possible at this stage to find what determines the iteration of each loop and how many times it will execute without additional analysis. Therefore, the formula is used as an approximate value. One use envisioned for the multi-microprocessor machine is to do

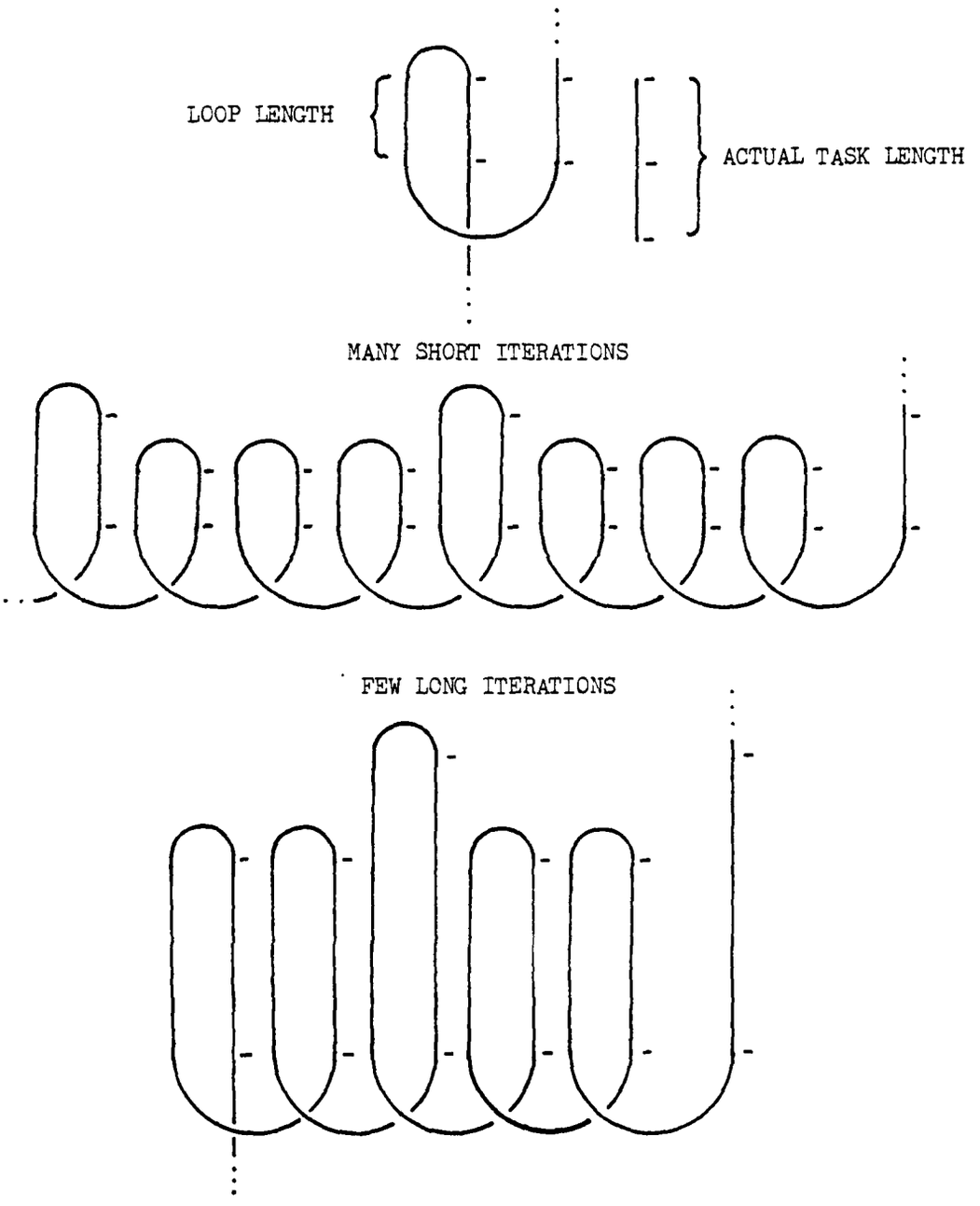


Figure 15

Length of Nested Loops

experimental program executions to find the best possible schedule or processor allocation. When this is found, the program would be considered ready to run on a production basis. One further comment is necessary.

Although it is not highly significant here to study the bounds on SF, it should be noted that it is usually positive. But it cannot be thought of as a positive number between zero and one. It approaches infinity as the number of instructions in loops approaches the total number of executable instructions. This is an undesirable situation indicating loops that are too long. Further research will be helpful in interpreting this factor for assembly language programs. For the remainder of this chapter all FORTRAN format statement labels referenced are shown in Appendix B. They were also shown in Figs. 7 through 14.

Possible Errors Noted by the Analysis

Possible errors noted by the analysis are conditions which may be due to faulty logic in the loop or jump structure of the source program. They could cause problems in execution. They are explained here in order of the format label number. Statement 9001,"NO EXIT FROM BACK JUMP, UNLESS BY RETURN OR OVERLAPPING LOOP," means an endless loop. Analysis of several programs showed that this is not unusual for microprocessor systems, because they are often designed to run a program over and over. Therefore, this is shown as a possible error. Statement 9902, "NO ENTRY TO LOOP, UNLESS BY RETURN FROM JUMP OUT," means that the loop will not be executed. This indicates a backward jump that appears to have been circumvented by a previous jump. This message may be

generated erroneously by jumps to another part of memory that has a return mechanism that is not discernable by the analysis.

Statement 9493, "NO PATH TO INSTRUCTION EXCEPT BY CALL," means there is at least one instruction which will not be executed because of a previous unconditional jump. This message may be generated erroneously by some programs that include vectoring or some sequence of unconditional jumps with no apparent return.

Statement 9993, "UNCONDITIONAL FORWARD JUMP AROUND FORWARD JUMP," is nearly the same as 9493 and in some cases confirms it. 9493 is generated during the instruction scan, and 9993 is generated by checking the jump table to confirm that there is no apparent path.

Warnings Made by the Analysis

Warnings are for conditions that have occurred during the analysis that will cause the results to be incorrect. With one exception these should not occur unless the source program size limits have been exceeded. These warnings are discussed in the order of the format statement numbers.

Statement 9000, "LOOP NUMBER FOR BACK JUMP NOT FOUND," means the loop was not recognized and stored in the loop table. This error should not occur unless accompanied by 9991 or 9996. If it does occur alone, it means there is a fault in the analysis program.

Statement 9991, "ARRAY LP OVERFILLED MORE THAN 100 LOOPS." This is self explanatory and means the array must be enlarged to accomodate the subject program.

Statement 9996, "ARRAY JP OVERFILLED MORE THAN 200 JUMPS." This is self explanatory and means the array must be enlarged to accomodate the subject program.

Statement 9980, "LOOP OVERLAPPED. SF VALUE MAY NOT BE MEANINGFUL." This means that two loops are overlapped. A loop jumps out of a subsequent loop which jumps back into the former one as was shown in Fig. 10. Unless these two loops are both nested in a third loop the LFPG will be in error. The analysis could not accomodate two nodes which partially converge or are not discrete and distinct. If the overlapped loops are nested in another, all three will be treated as one task for partitioning, and the overlap will be inconsequential. This version of the program does not make this determination, so all overlapped loops generate the warning.

Statement 9999, "UNCLASSIFIED JUMP," means the jump type or loop class was not established. It indicates an array problem or fundamental error in the suitability checker. This error should not occur.

Notes Made by the Analysis

Notes are for conditions discovered in the source program that are not necessarily wrong but considered essential to emphasize. They may indicate a problem, but will be informative in any case. They are discussed in order of the format statement number as shown in Appendix B.

Statement 9490, "FWD JUMP IS NOT ASSOCIATED WITH ANY LOOP," means an isolated forward jump. These are detrimental to parallel processing, especially if conditional.

Statement 9903, "CONDITIONAL FWD JUMP ENTERS BACK JUMP FROM OUTSIDE ITS RANGE." This could be an error, or it may simply be a way of entering a loop.

Statement 9990, "JUMP ALSO JUMPED AROUND JUMP." This indicates that a forward jump went around another jump subsequent to the one shown in the jump table. It is not necessarily a problem. It supplements the jump analysis, because the association can only be stored for one jump.

Statement 9994, "FWD JUMP AROUND BACK JUMP." This shows a jump around a loop. It may indicate a problem, if there is no other path to the loop. This is a precautionary note, because all information was not available at the time to make an unqualified error identification.

Statement 9995, "CONDITIONAL FWD JUMP IN BACK JUMP, BUT IS NOT SIGNIFICANT TO ITS STRUCTURAL CLASSIFICATION." This indicates an interior jump has been found that is not a minimum requirement of loop structure for classification purposes.

Statement 9997, "FORWARD JUMP ALSO ASSOCIATED WITH JUMP." This indicates a jump out of an inner loop of nested loops. According to the analysis it would be associated with all loops it jumps out of. However, only one association can be made in the jump table.

This concludes discussion of the available diagnostic factors, errors, warnings, and notes. An example program listing showing some of them is included in Fig. 17 in Chapter V, which discusses actual experimental results.

V. EXPERIMENTAL RESULTS

This chapter shows actual results achieved by using the suitability checker on ten microprocessor programs. The first section discusses these results and the second section explains the output of a sample program run.

Findings

Eight actual INTEL 8080 programs and two pseudo programs (two test cases written for this research) were analyzed for parallel processing suitability. The results of applying the suitability checker to these programs are shown in Fig 16. Probably the most significant finding was that most of the programs were suitable except those containing overlapped loops which were not nested in another loop. The one program, NONDIGIT, which was unsuitable, because of its long outside loop structure, was found to be highly suitable when the outer loop was removed. In one other case, TEST P2411, when the outer loop was removed, this exposed overlapped loops rendering the program unsuitable in that context. Programs containing exposed overlapped loops are shown with an asterisk.

Also the results are shown in parenthesis for an earlier version that neglected to discount nested loops from the loop count. Although this did make a difference, it was almost negligible for the programs tested.

PROGRAM NAME	PURPOSE	SF	I/O	ARITH LOG	CALLS	OUTER LOOPS	UNASSOC FWD JUMP	INSTR IN LOOPS	TOTAL INSTR
ADD32	ADD TWO 32 DIGIT NUMBERS	.79	0	8	0	1	0	7	14
DUMP	DUMP MEMORY BETWEEN TWO SPECIFIED LOCATIONS	(0.65)* 0.63*	0	31	19	(4) 3	1	59	102
FPPKG	FLOATING POINT ARITH. PACKAGE	0.30*	0	178	127	26	22	470	825
HEXNUM 8080	INPUTS 4 DIGIT HEX NUMBERS	(0.66) 0.62	0	30	19	(5) 3	1	36	85
LOADER	FILM DATA LOADER	(0.26) 0.18	8	16	14	(6) 2	9	41	90
MCHEC	MEMORY DIAGNOSTIC	0.35*	1	38	19	4	5	76	140
MDSMCHEC	CHECK MEMORY BY WRITING & READING NUMERIC PATTERNS	0.37 0.35	0	35	45	(5) 3	4	53	173
NONDIGIT	CONTROLLER FOR FILM READING MACHINE	** <2.98>	14	41	15	1 <4>	0 <13>	223 <203>	223 <222>
LOOPTEST	TEST CASE FOR LOOP RECOGNITION	(13.05) 7.05 [10.06]*	0	7	0	(7) 1 [4]	0	18 [17]	19 [18]
TEST P2411	TEST CASE FOR NESTED LOOPS	(0.83) 0.74	0	41	0	(16) 13	8	63	96

* exposed overlapped loops ** no data [ ] outer loop removed

< > SF could be measured only if outer loop removed ( ) nested loops not discounted

Figure 16

Experimental Results

It is significant that the checker can be used for parts of programs. Although assembler errors will be generated, the analysis will still be completed. The next section tells about the results shown on the program listing.

Explanation of a Sample Program Output

This discussion deals with the listing shown in Fig. 17. The entire figure is nine pages. The format of this output is the jump analysis followed by the INTEL 8080 source program listing and symbol table. The jump analysis was output first, because it reveals information about the source list. The source list must be referenced, however, to use the analysis. This reference is made through the jump numbers which are shown in the jump table and on the source list under the column label, JUMP. See Fig. 17 (cont.) on the fourth page. Both jump numbers and loop numbers are given in the analysis.

The analysis format begins with the errors, notes, and warnings generated by the loop construct analysis. This particular example had no warnings, mainly because there were no overlapped loops. This part is followed by nesting information from the nest check. Then the suitability factor is shown, which is 0.35 in this example. The values of variables used to calculate SF are shown over the top of the jump table. See Fig. 17.

The jump table format is the same as explained previously for array JP. The loop numbers for backward jumps have been added to the left side. They are followed from left to right by the jump number, its executable instruction number (both shown on the source list), the executable instruction number of the destination, the forward jump type

```
--- INTEL 8181 JUMP ANALYSIS --- VER 2.00, 31 DEC 79 ---

*** POSSIBLE ERROR, NO EXIT FROM BACK JUMP   13 (LOOP   5)
    UNLESS BY RETURN INSTRUCTION OR OVERLAPPED LOOP.

NOTE, GOOD FWD JUMP    9 IN BACK JUMP   11 (LOOP 4),
    BUT IS NOT SIGNIFICANT TO ITS STRUCTURAL CLASSIFICATION.

NOTE, GOOD FWD JUMP    8 IN BACK JUMP   11 (LOOP 4),
    BUT IS NOT SIGNIFICANT TO ITS STRUCTURAL CLASSIFICATION.

*** POSSIBLE ERROR, NO EXIT FROM BACK JUMP    7 (LOOP   3)
    UNLESS BY RETURN INSTRUCTION OR OVERLAPPED LOOP.

NOTE, END JUMP    5 ALSO ASSOC WITH JUMP    7.

NOTE, END JUMP    4 ALSO ASSOC WITH JUMP    7.

NOTE, END JUMP    2 ALSO ASSOC WITH JUMP    7.

NOTE, END JUMP    1 ALSO ASSOC WITH JUMP    7.

NOTE, LOOP    2 NESTED IN LOOP    3

NOTE, LOOP    1 NESTED IN LOOP    3

SUITABILITY FACTOR FOR PARALLEL PROCESSING IS      .36.
    ANY VALUE GREATER THAN 0.3 IS FAVORABLE.

                JUMP TABLE

   NUMBER   NIO   NAL   NC    NB    NF    NL    NX
       13     0    35   46     3     4    53   173

 LOOP  JUMP FROM   TO   TYPE ASSOC
          1    57    64     5     3
          2    61    64     6     3
          3    63    58     2     3
    1   **********************
          4    71    81     6     6
          5    77    81     6     6
          6    79    85     2     6
    2   **********************
          7    84    54     2     7
    3   **********************
          8   125   131     6    10
          9   131   137     8    10
         10   131   121     1    10
    4   **********************
         11   155   153     6   -12
         12   157   153     8   -12
         13   163   152     2    13
    5   **********************
```

Figure 17

Sample Program Output

```
                JUMP ANALYSIS

NOTE. FWD JUMP  12, EX INSTR NO. 157
     IS NOT ASSOCIATED WITH ANY LOOP.

        LOOPS IN THIS PROGRAM

LOOP #      FROM        TO        CLASS

 0001       0063        0155        2
        FORWARD JUMPS ASSOCIATED WITH LOOP   1

        JUMP  1 EX INSTR NO. 0157

        JUMP  2 EX INSTR NO. 0161

 0002       0074        0065        2
        FORWARD JUMPS ASSOCIATED WITH LOOP   2

        JUMP  4 EX INSTR NO. 0075

        JUMP  5 EX INSTR NO. 0077

 0003       0084        0054        2
        FORWARD JUMPS ASSOCIATED WITH LOOP   3

                    NONE

 0004       0135        0121        1
        FORWARD JUMPS ASSOCIATED WITH LOOP   4

                    NONE

 0005       0138        0102        2
        FORWARD JUMPS ASSOCIATED WITH LOOP   5

                    NONE

  --- INTEL 8080 CROSS ASSEMBLER --- V2.0 [ ] DEC 78 ---

 P C  MACH CODE  JUMP  LFPG  EX NO  LABEL  INST  OPERANDS

**************** BEGIN MDSMCHEC ************************
*ABSTRACT:                                               *
*       THIS PROGRAM PROMPTS THE USER TO ENTER TWO PAIRS*
*OF NUMBERS.  THE FIRST PAIR IS THE BEGINNING AND ENDING*
*ADDRESSES OF MEMORY TO BE CHECKED.  THE SECOND PAIR IS THE*
*TWO NUMBERS THAT WILL ALTERNATE IN FILLING MEMORY.  AFTER*
*MEMORY IS FILLED, THE PROGRAM WILL CHECK IT AGAINST THE*
*FILL NUMBERS AND OUTPUT ANY DIAGNOSTICS.  THEN THE TWO FILL*
*NUMBERS ARE SWAPPED AND THE SEQUENCE OF FILLING, CHECKING,*
*AND SWAPPING IS REPEATED CONTINUOUSLY.                  *
*                                                       *
*CALLING SEQUENCE/PARAMETERS:                           *
*                                                       *
*       PROGRAM PROMPTS: ENTER LO,HI OF MEMORY TO BE CHECKED*
*       USER RESPONDS:   XXXX,XXXX     (X=HEX DIGIT)    *
*                                                       *
*       PROGRAM PROMPTS: ENTER NUM1,NUM2  (TWO FILL NUMBERS)*
*       USER RESPONDS:   XX,XX         (X=HEX DIGIT)    *
*                                                       *
*       PROGRAM RESPONDS: DIAGNOSTICS                   *
*                         ADDRESS  NUM  MEMORY          *
*                         XXXX     XX   XX              *
*********************************************************
```

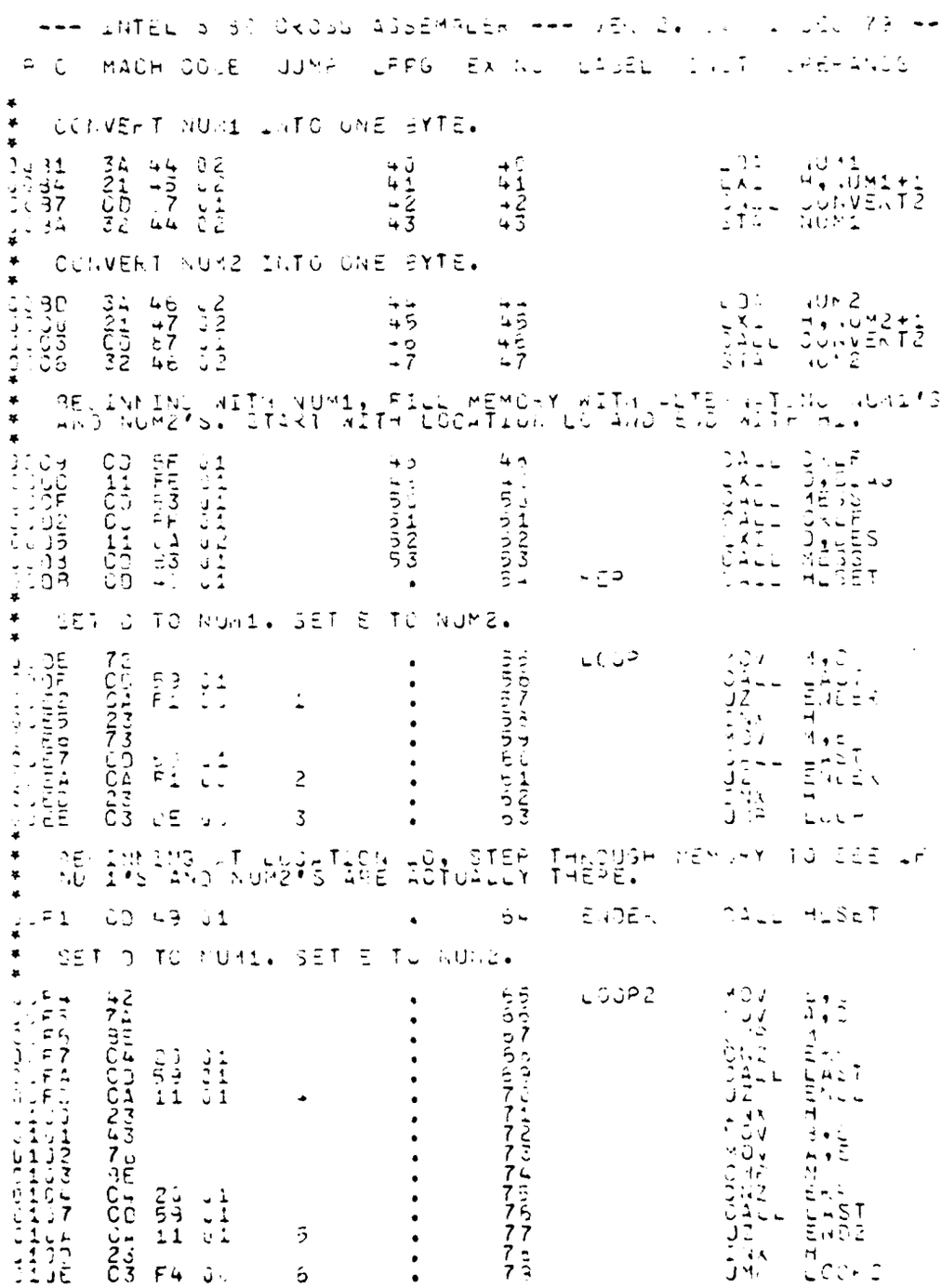
Figure 17 (continued)

Sample Program Output

```
--- INTEL 8080 CROSS ASSEMBLER --- VER 2.0, 01 DEC 78 ---
 LOC  MACH CODE   JUMP  LFFG  EX NO  LABEL   INST  OPERANDS
F803                                 CI      EQU   X'F803'
*
F809                                 CO      EQU   X'F809'
*
0040  31 43 02           1      1    START   LXI   SP,STACK
0043  CD 8F 01           2      2            CALL  CRLF
*
*  WRITE PROMPT FOR LO AND HI
*
0046  11 CA 01           3      3            LXI   D,PMT1
0049  CD 63 01           4      4            CALL  MESS
004C  CD 8F 01           5      5            CALL  CRLF
*
* INPUT LO AND HI AS FOUR BYTES EACH(ASCII CODE)
*
004F  11 27 02           6      6            LXI   D,LO
0052  CD 63 01           7      7            CALL  CONVERT1
0055  0E 20              8      8            MVI   C,X'20'
0057  CD 09 F8           9      9            CALL  CO
005A  11 2B 02          10     10            LXI   D,HI
005D  CD 63 01          11     11            CALL  CONVERT1
*
* CONVERT LO AND HI TO LOW-HIGH MEMORY ADDRESSES.
*
0060  3A 27 02          12     12            LDA   LO
0063  21 28 02          13     13            LXI   H,LO+1
0066  CD 87 01          14     14            CALL  CONVERT2
0069  32 27 02          15     15            STA   LO
006C  3A 29 02          16     16            LDA   LO+2
006F  21 2A 02          17     17            LXI   H,LO+3
0072  CD 87 01          18     18            CALL  CONVERT2
0075  32 28 02          19     19            STA   LO+1
0078  3A 2B 02          20     20            LDA   HI
007B  21 2C 02          21     21            LXI   H,HI+1
007E  CD 87 01          22     22            CALL  CONVERT2
0081  32 2B 02          23     23            STA   HI
*
0084  3A 2D 02          24     24            LDA   HI+2
0087  21 2E 02          25     25            LXI   H,HI+3
008A  CD 87 01          26     26            CALL  CONVERT2
008D  32 2C 02          27     27            STA   HI+1
*
*  WRITE PROMPT FOR NUM1,NUM2
*
0090  CD 8F 01          28     28            CALL  CRLF
0093  11 EE 01          29     29            LXI   D,PMT2
0096  CD 63 01          30     30            CALL  MESS
0099  CD 8F 01          31     31            CALL  CRLF
*
* INPUT NUM1 AND NUM2 AS TWO BYTES EACH (ASCII CODE)
*
009C  11 44 02          32     32            LXI   D,NUM1
009F  06 02             33     33            MVI   B,C'2'
00A1  CD 65 01          34     34            CALL  NEXT
00A4  0E 20             35     35            MVI   C,X'20'
00A6  CD 09 F8          36     36            CALL  CO
00A9  11 46 02          37     37            LXI   D,NUM2
00AC  06 02             38     38            MVI   B,C'2'
00AE  CD 65 01          39     39            CALL  NEXT
```

Figure 17 (continued)

Sample Program Output



```
--- INTEL 8 80 CROSS ASSEMBLER --- VER. 2.  .  . DEC 79 ---
P C  MACH CODE  JUMP  LPRG  EX NO  LABEL  INST  PRE-ANOS

*
*   CONVERT NUM1 INTO ONE BYTE.
*
                                             MOV   A,M
                                             ANA   A
                                             JM    CONVERT2
                                             STA   NUM1
*
*   CONVERT NUM2 INTO ONE BYTE.
*
                                             MOV   A,M
                                             ANA   A
                                             JM    CONVERT2
                                             STA   NUM2
*
*   BEGINNING WITH NUM1, FILL MEMORY WITH CONSECUTIVE NUM1'S
*   AND NUM2'S.  START WITH LOCATION 0C0 AND END WITH 4..
*
                                             LXI   H,0C0H
                                             MVI   A,4
                                             ADD   M
                                             INX   H
                                             MOV   M,A
                                             CPI   S
                                             JNZ   LOOP
                                             HLT
*
*   SET C TO NUM1. SET E TO NUM2.
*
                                             LOOP  LDA   NUM1
                                             MOV   C,A
                                             LDA   NUM2
                                             MOV   E,A
*
*   BEGINNING AT LOCATION 0C0, STEP THROUGH MEMORY TO SEE IF
*   NO 1'S AND NUM2'S ARE ACTUALLY THERE.
*
                                             ENDER CALL  HLSET
*
*   SET D TO NUM1. SET E TO NUM2.
*
                                             LOOP2 LDA   NUM1
                                             MOV   D,A
                                             LDA   NUM2
                                             MOV   E,A
                                             CALL  HLSET
                                             DCR   D
                                             JNZ   LOOP2
```

Figure 17 (continued)

Sample Program Output

or loop class, and associated jump. For this example jump 1 goes from executable instruction number 57 to 64. It is a type 6 which is conditional forward as confirmed by the source list where it is shown as JZ ENDER. This jump is associated with jump 3 which is also loop 1 at executable instruction 63. The label ENDER is shown at executable instruction number 64. Therefore, jump 1 jumps out of loop 1. Also, as the analysis shows, jump 1 is associated with jump 7 (loop 3), because loop 1 is nested in loop 3. Therefore, jump 1 is part of both loops, although the jump table only shows the association with the inner loop, because of the way the table is constructed. At the bottom of the table, notes and errors derived from the final jump table analysis are shown. In this case the only note was that jump 12 is an isolated forward jump which is not conducive to parallel processing. Following this, the analysis concludes by listing the loops and their associated jumps. This is the same information as the jump table, but it is arranged with respect to loops rather than jumps. Also note that jumps such as 8 and 9 are not shown as associated with the loop, because they are not significant to the structural classification of the loop containing them, loop 4. In other words, they have no influence on program flow into or out of the loop. Therefore, although the loop list might be considered redundant, it is interpreted in a slightly different manner, for convenience in determining loop structures. This was done for possible later use in analyzing loop iterations. It would be easier to decide which jumps to examine, given that some of them are not structurally significant. Using all analysis information assists in examining the source list.

```
--- INTEL 8080 CROSS ASSEMBLER --- VER 2.0, 31 DEC 79 ---
P C  MACH CODE  JUMP  LPPS  EXTNL  LABEL  INST  OPERANDS

*
*  SWAP NUM1 AND NUM2. THEN DO IT AGAIN.
*
0111  3A 44 02              .      83      EX02      LDA   NUM1
0114  2A 46 02              .      84                SHLD  NUM2
0117  32 46 02              .      85                STA   NUM2
011A  22 44 02              .      86                SHLD  NUM1
011D  C3 09 00     7        54     54                JMP   REP
*
*  DIAGNOSTIC DETECTED  ******
*
0120  CD BF 01              55     55      EXX       CALL  CRLF
0123  CD 3E 01              56     56                CALL  ADR
0126  11 00 02              57     57                LXI   D,BLK5
0129  CD 63 01              58     58                CALL  MESS
012C  78                    59     59                MOV   A,B
012D  CD 50 01              60     60                CALL  CONVERT3
0130  11 23 02              61     61                LXI   D,BLK3
0133  CD 63 01              62     62                CALL  MESS
0136  7E                    63     63                MOV   A,M
0137  CD 50 01              64     64                CALL  CONVERT3
013A  CD 56 01              65     65                CALL  DESET
013D  C9                    66     66                RET
*
*  OUTPUT ADDRESS NUMBER
*
013E  AF                    67     67      ADR       XRA   A
013F  BC                    68     68                CMP   H
0140  CD 50 01              69     69                CALL  CONVERT3
0143  AF                    70     70                XRA   A
0144  BD                    71     71                CMP   L
0145  CD 50 01              72     72                CALL  CONVERT3
0148  C9                    73     73                RET
*
*  POINT H+L TO MEMORY
*
0149  21 27 02              74     74      HLSET     LXI   H,LC
014C  5E                    75     75                MOV   E,M
014D  23                    76     76                INX   H
014E  56                    77     77                MOV   D,M
014F  EB                    78     78                XCHG
0150  3A 46 02              79     79      DESET     LDA   NUM2
0153  BE                    80     80                CMP   M
0154  3A 44 02              81     81                LDA   NUM1
0157  57                    82     82                MOV   D,A
0158  C9                    83     83                RET
*
*  COMPARE CURRENT ADDRESS TO HI.
*
0159  3A 28 02              84     84      LAST      LDA   HI
015C  BC                    85     85                CMP   H
015D  C2                    86     86                JNZ   RET
015E  3A 2C 02              87     87                LDA   HI+1
0161  BD                    88     88                CMP   L
0162  C9                    89     89                RET
*
*  CONVERT ASCII INPUT TO BINARY EQUIVALENT.
*
0163  06 04                 90     90      CONVERT3  MVI   B,4
0165  CD C3 FB              .      91      NEXT      CALL  GETCH
0168  E6 7F                .      92                ANI   X'7F'
```

Figure 17 (continued)

Sample Program Output

```
   --- INTEL 8080 CROSS ASSEMBLER --- V2.0, 31 DEC 79 ---

 P C   MACH CODE   JUMP  LEFT    EX NO   LABEL   INST   OPERANDS

 016A   32 49 02                  123             STA   TEMP2
 016D   4F                        124             MOV   C,A
 016E   CD C9 F8                  125             CALL  TOUT
 0171   3A 49 02                  126             LDA   TEMP2
 0174   FE 3A                     127             CPI   X'3A'
 0176   F2 7E 01     8            126             JP    X1

*  CHARACTER IS FROM 0 TO 9.

 0179   D6 30                     129             SUI   X'30'
 017B   C3 81 01     8            130             JMP   XX1

*  CHARACTER IS FROM A TO F.

 017E   D6 37                     131     X1      SUI   X'37'
 0180   12                        132     XX1     STAX  D
 0181   13                        133             INX   D
 0182   05                        134             DCR   B
 0183   C2 65 01    10            135             JNZ   NEXT
 0186   C9                        136             RET

*  PACK TWO BYTES INTO ONE BYTE.

 0187   07                        137     CONVERT2 RLC
 0188   07                        138             RLC
 0189   07                        139             RLC
 018A   07                        140             RLC
 018B   B6                        141             ORA   M
 018C   C9                        142             RET

*  UNPACK ONE BYTE INTO TWO BYTES. EACH BYTE WILL BE A VALUE
*  FROM 0 TO F. THEN CONVERT EACH BYTE TO ASCII FOR OUTPUT.

 018D   32 49 02                  143     CONVERT3 STA   TEMP
 0190   0F                        144             RRC
 0191   0F                        145             RRC
 0192   0F                        146             RRC
 0193   0F                        147             RRC
 0194   E6 0F                     148             ANI   X'0F'
 0196   CD A2 01                  149             CALL  CONVERT4
 0199   3A 49 02                  150             LDA   TEMP
 019C   E6 0F                     151             ANI   X'0F'
 019E   CD A2 01                  152             CALL  CONVERT4
 01A1   C9                        153             RET

*  CONVERT BINARY TO ASCII AND OUTPUT.

 01A2   FE 0A                     154     CONVERT4 CPI   X'A'
 01A4   F2 AC 01    11            155             JP    X2

*  CHARACTER IS FROM 0 TO 9.

 01A7   C6 30                     156             ADI   X'30'
 01A9   C3 AE 01    12            157             JMP   XX2

*  CHARACTER IS FROM A TO F.

 01AC   C6 37                     158     X2      ADI   X'37'
 01AE   4F                        159     XX2     MOV   C,A
 01AF   CD C9 F8                  160             CALL  TOUT
 01B2   C9                        161             RET
```

Figure 17 (continued)

Sample Program Output

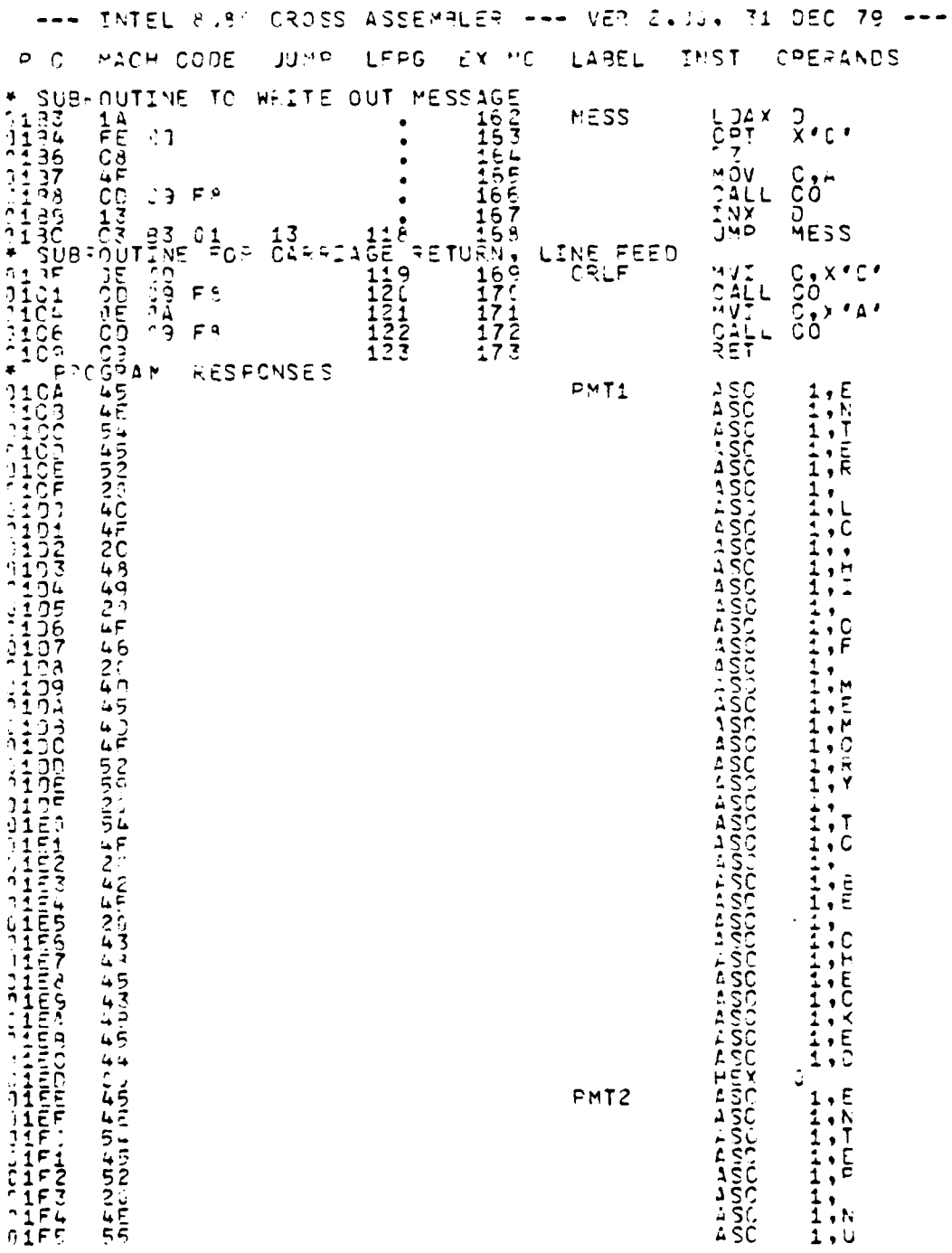


Figure 17 (continued)

Sample Program Output

The source list closely resembles an ordinary assembly language listing except for the addition of the columns for JUMP, LFPG, and EX NO in the middle of the page. This example listing in Fig. 17 was slightly modified to fit the margins by deleting the line numbers which would be on the far left and bringing some comments from the far right to the far left. Also, it should be noted that the symbol table is unchanged from the original. See Fig. 17 (cont.).

In designing the program output, consideration was given to using the program address counter or line numbers for analysis reference rather than executable instruction numbers. Although the executable instruction numbers had to be generated in the program, they were easily added and are much simpler to use and reference than either line numbers or program addresses. Line numbers are not definitive enough, because they include comments and pseudo ops that have no bearing on the analysis. Program addresses are harder to work with, because they are hexadecimal. Therefore, executable instruction numbers were used and shown for the listing to clarify the analysis.

The final product of the analysis is the nodes of the LFPG which are shown on the source list between jump number and executable instruction number. These numbers can be considered a map of the nodes of the LFPG of the microprocessor source program. The only part missing from the graph is the edges or arrows between nodes. These can easily be determined by looking back into the jump table where they are shown in the FROM and TO columns. By looking at the LFPG nodes it is easy to see the outer loops which are considered as a single node or task. These loops are shown by dots instead of numbers to indicate a series of

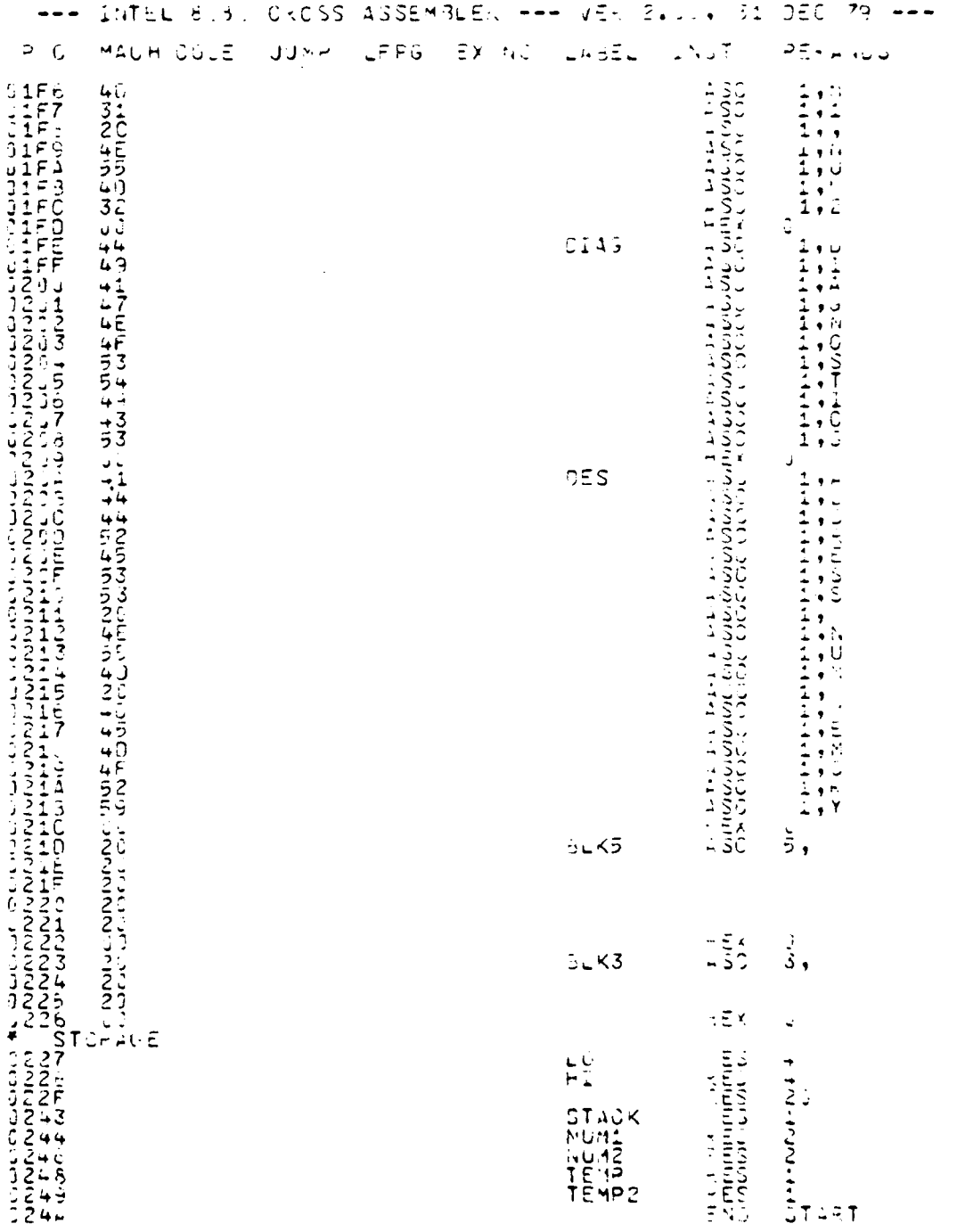


Figure 17 (continued)

Sample Program Output

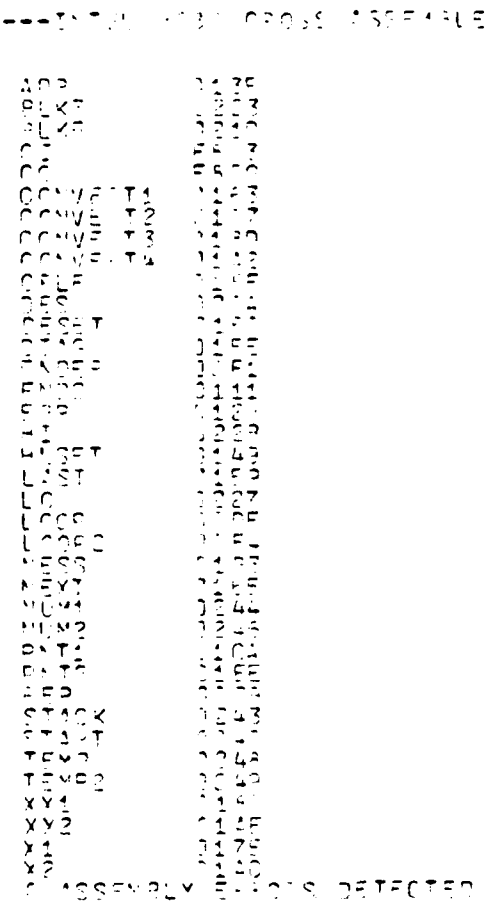


Figure 17 (continued)

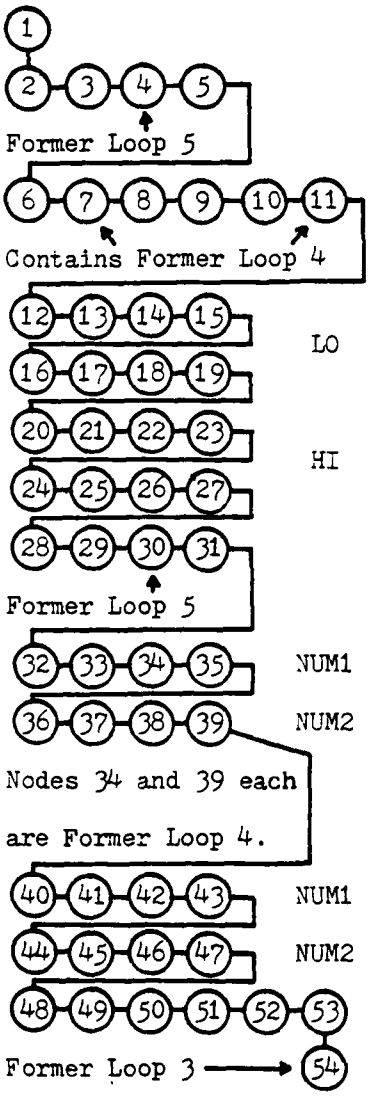
Sample Program Output

instructions in the loop considered as one task. The node denoting the loop is the number corresponding to the backward jump that forms the loop. This information represented by the LFPG could be used as input for a slightly modified parallel task recognizer as discussed previously.

For clarity the actual LFPG is shown in Fig. 18. To be used with the parallel task recognizer, this graph would be used with task transition information to construct the parallel processable task graph as explained in Chapter II. There are only 54 nodes in this graph because nodes 55 to 123 are actually parts of subroutines as shown on the output list in Fig. 17. Note that loops four and five are contained in these subroutines. Therefore, they occur more than once. This indicates that it would be desirable to build a table of calls and returns to cross reference with the jump table for finding loops nested due to calls. This would also improve error diagnostics as mentioned in the next chapter.

Other specific observations relate to this particular program. It points out the fact that the suitability checker is limited to recognizing input and output by the IN and OUT instructions. It is obvious that this program has a good balance of I/O and internal operations. However, the I/O is done by subroutines (in the read only memory) which are known to this program only as CALL CI or CALL CO. This would appear to be a positive factor for parallel processing in this program that is not discernable by this suitability checker. But there are negative factors that are subtle also. The internal operations are too closely associated with the I/O, because the checking

NOTE: All node numbers refer to Fig. 17 LFPG numbers.



FUNCTION

INTERNAL - Load Stack Pointer.

OUTPUT - Prompt user to enter HI and LO limits for memory check.

INPUT - Receive HI and LO as four bytes.

INTERNAL - Convert LO and HI to machine language memory addresses.

OUTPUT - Prompt user to enter NUM1 and NUM2 for memory check.

INPUT - Receive the numbers as two bytes each.

INTERNAL - Convert each number to one byte.

OUTPUT - Display header.

INTERNAL - Fill memory with alternating NUM1 and NUM2 from LO to HI, check, exchange NUM1 and NUM2, and repeat.

Figure 18

Loop Free Graph of the Sample Program

done by the process in node 54 used the same variable names for HI and LO as the input portion. This is an obvious conflict that could be eliminated by using a buffer, if the programmer had been thinking in terms of parallel processing.

Although it is necessary to use the parallel task recognizer to find the optimum partitions, it is obvious that the program could be simply partitioned between nodes 53 and 54. That is, one processor could do the I/O and preparation while another checked the memory. There is a possible conflict between the I/O and the ERR routine which uses the MESS and CRLF routines as shown in Fig. 17. This would happen if an error were discovered, which would direct a diagnostic message. This could be overcome by letting the memory checking processor interrupt the other processor to perform the diagnostic message as discussed in Appendix A.

Another point to make is, that for parallel processing, it might have been better for the initial prompt to have requested all necessary information which could have then been processed in parallel. This clearly shows, that the programmer cannot be disregarded. Programs written for uniprocessors will necessarily be limited in parallelisms by their structure. But improvements can be made by processing portions of these programs in parallel. However, one of the most needed improvements is to emphasize the need to program for parallelism rather than sequential processing. The next chapter discusses conclusions and recommendations.

VI. CONCLUSIONS AND RECOMMENDATIONS

As the concluding chapter, this discussion will emphasize the significance of the work, general findings, suggested complementary work, and use of the multi-microprocessor system.

The objective of this thesis was to incorporate suitability checking into a cross assembler as a step toward the goal of assembler scheduled parallel processable program partitions for a multi-microprocessor system. This objective was achieved by solving the major problem of detecting loops in the microprocessor assembly language source program and adapting Ramamoorthy's suitability checker for use with assembly language. The additional diagnostics for structural analysis of the source program were an additional benefit incidental to the problem solution.

Significance of This Work

The research reprorted within has important implications for further research and for debugging existing or developing programs. The suitability factor, SF, enables an easy meaningful estimate of potential for parallel processability of the subject assembly language source program. This gives ready indication of which programs should be subjected to further refinement such that they may be efficiently executed on a multi-microprocessor system. It enables finding the nodes of Ramamoorthy's reduced program graph or loop free program graph (LFPG)

[13]. This is a task model of the subject program where all outside loops are considered as a single task. The actual graph can be constructed from information available on the output listing of the analysis.

The analysis of an assembly language source program does not require any of the large square matrices used by Ramamoorthy's parallel task recognizer before having some assurance that there is indeed potential for parallelism. The arrays that it does use are not large relative to a medium sized computer system. See Appendix C. They could be compressed by superimposing some arrays on others, but the saving to be realized is not deemed worthwhile compared to the increased complexity that would be required. The presented version actually combines the capability of a suitability checker and Phase I of Ramamoorthy's parallel task recognizer [14]. The information from the LFPG could be combined with applicable task transition information and subjected to Ramamoorthy's parallel task recognizer to obtain the task partitions necessary for parallel processing.

Ramamoorthy's suitability checker program differs from this one with respect to loops in two ways. His program did not allow nested loops and did not recognize loops created by backward jumps or branches. It dealt only with DO loops [11]. This program recognizes three classes of FORTRAN-like loops in the assembly language program as well as nested loops. It finds and notes overlapped loops which may be detrimental to parallel processing. The ability to recognize loops will aid further research in the area of determining program segments which may be executed in parallel on a multi-microprocessor system.

Diagnostic messages output with the jump analysis and source list have never been available before for microprocessor assembly language programs to find endless loops, find loops with no entry, find nesting errors, and analyze program flow or structure. Although the diagnostics have some conditions associated with them, they will prove generally useful for debugging or analyzing programs. Programs that use "vectoring" will generate erroneous error messages, because this analysis does not trace the vector. It would be possible to improve the diagnostic capability by building a table of calls and returns for tracing flow vectors and giving more definitive error messages. Error messages for loops with no exit must be regarded with program design in mind, because many microprocessor program designs purposely include endless loops due to their dedicated nature. These diagnostics actually tell more about how the program is structured than whether it is logically correct. But they do provide objective automated analysis, and will point out some problems. Using the analysis has already revealed some general conclusions.

General Conclusions

The suitability of an assembly language source program for parallel processing still depends on the programmer to a large extent. This was readily shown by the sample program in Figs. 17 and 18. Even when using implicit parallelisms the programmer cannot be disregarded. The same process coded in different ways will have varying potential for parallel processing. Using this suitability checker could help develop guidelines by evaluating different approaches and selecting the best one. As a software design tool, this suitability checker could be used

to encourage programmers to look for ways to facilitate parallel processing and displace the habit of sequential programming. Some initial guidelines follow.

Of the programs analyzed, those which had the best suitability factors had simple closed loops, often with no exit. This supports Ramamoorthy's findings that complex decision structures are not conducive to parallel processing. Conversely, a system with a simple loop could be set up to use one processor for an input/output driver while another handled interrupts or did calculations.

However, research showed that some programs which had a poor suitability rating could be broken up into subportions which showed good potential for parallel processing. This was especially true for programs which were written as one large loop where the back jump was located at or near the end, and the return was at or near the beginning. If the program is one large loop, the SF cannot be measured except below the level of the loop. See Fig. 16. This is because the loop is one task, and the analysis will yield a SF approaching infinity as NX - NL = 0. Also if the program is one large loop, some problems of parallel processing are masked. Forward jumps not really associated with loops will not be recognized as such, because they are contained in the large loop. This suggests guidance for using loops in assembly language programs.

To facilitate parallel processing, programs written with a main loop should be designed to balance the loop with other tasks. Put only necessary code in the loop, so it is not too long. Any task or process that can be put in a subroutine can probably be handled by another

processor. Also, if any overlapped loops are used, try to nest them in another loop. More guidance of this type could be gained by analyzing more programs using the suitability checker. Many questions, as outlined below, still need to be answered.

Suggested Complementary Work

One of the most important questions remaining to be answered is how to implement the remainder of the parallel task recognizer for assembly language programs. This could be done in an interim pass between assembler passes one and two. It should not be necessary to change Ramamoorthy's recognizer significantly for this. The most difficult task seems to be that of automating the analysis of task transitions necessary to develop the parallel processable task graph for the assembly language program. This was explained in Fig. 1. This requires finding the links between each task and the next task that uses memory used by that task [12,14]. When this has been done the task transition graph and the loop free program graph could be used with Ramamoorthy's parallel task recognizer [11].

The next question remaining to be answered is how to load the parallel processable portions of the program into separate memories with synchronization primitives as aids to interprocessor communication and scheduling. These would need to be based on the earliest and latest task scheduling times derived from the parallel task recognizer for each program partition [14]. The assembler could append these primitives to each code partition and configure the load for two separate memories by using two program counters. Some trial and error balancing might be required to optimize the process allocations to each memory. This would

be important in the development of a multi-microprocessor system. The schedule or processor assignment must be tested and optimized for production execution.

Scheduling Considerations

Lorin described a streamlined machine with specialized individual processors for scheduling; execution; and load, store, modify, or transfer operations [10]. However, such a system would require breaking assembly language instructions down into fragments at the microprogram level. Also it would require that each processor have the capability to directly load and manipulate registers in the other processors. This might be advantageous for a dedicated design requiring very high speeds. But it would require detailed analysis and many design compromises without the probable return on investment of a more generalized system using standard parts and having more possible applications.

A simplified version of a dynamically scheduled system could be built using standard processors with shared memories. This was the initial attempt at solving the problem. Initial research showed that no particular machine is better suited than any other. However, the NATIONAL SEMICONDUCTOR family of IMP, PACE, and SC/MP machines looked promising, because they have built in control signals allowing "daisy-chaining" interrupt type communications. Also the IMP is microprogrammable and uses the same instruction set as the PACE. Therefore, an investigation of the scheduling problem was made for the PACE, so that the results could be applied to microprogrammable machines.

The results of this are shown in Fig. 19. Even by searching on a priority basis for the most often used instructions first, it will take from three to ten instruction cycles to recognize the instruction and jump to a routine that can handle it. Even using a more sophisticated sort would take two or three cycles. Therefore, a microprocessor operating as the scheduling unit for an application program would become hopelessly bogged down as it tried to assign individual instructions to the other two processors for preparation and execution. This would create a serious bottleneck at the outset without even considering the problems of memory contention, processor communication, and synchronization. After this initial dead-end in the research, it became apparent that a different approach would be necessary.

It seemed that the dynamic approach was not only too slow, but that there was too much overhead at execution time due to all the instruction fragmentation and transfers to different processors. Reflection on this problem led to the conclusion that if all this overhead could be accomplished beforehand, operation would be more efficient. The only logical time to do it was before the load. This left the assembly process as the only possible time to do the scheduling. This seemed compatible with Ramamoorthy's work [1,11,13,14], because it would be easier to schedule and synchronize partitions or blocks of instructions with less overhead than it would for fragments of individual instructions. It was only a question of how to find the parallel processable partitions, their execution time relationship, and the necessary primitives to synchronize them. Therefore, the problem inves-

```
*SEQUENCE FOR PRIORITY DECODE OF PACE INSTRUCTION SET

*R2 IS TRTP BECAUSE ONLY E2 OR E3 CAN LOAD INDIRECT

LOAD: LD    R3,(R2) *LOAD REG 3
      SKG   4FFF    *MAY BE IN E1
      JMP   PATH1   *L.T. E1
      SKG   7BFF    *NOT E1
      JMP   SNE1    *E1
      SKG   DFFF    *E2
      JMP   PATH2   *L.T. E2
      JMP   SNE2    *E2
*
PATH1:SKG   1FFF    *MAY BE E4
      JMP   PATH3   *L.T. E4
      SKG   3FFF    *T1
      JMP   SNE4    *E4
      JMP   TRNS1   *T1
*
PATH2:SKG   B7FF    *MAY BE E3, T4, NOT T3
      JMP   PATH4   *T3 OR STI
      SKG   C3FF    *G.T. E3
      JMP   SNE3    *E3
      SKG   A3FF    *MAY BE E6
      JMP   PATH5   *L.T. E6
      SKG   AFFF    *G.T. E5
      JMP   SENDE   *E6
      SKG   B7FF    *MAY BE E8
      JMP   SNDLT   *L3
      SKG   BBFF    *G.T. E8, MAY BE L1
      JMP   SENDE   *E8
      JMP   SNDL1   *L1
*
PATH3:SKG   07FF    *MAY BE E5
      JMP   STOP    *HALT
      SKG   13FF    *NOT E5, MAY BE T2
      JMP   SENDE   *E5
      JMP   TRNS2   *T2
*
PATH4:SKG   7FFF    *T3
      JMP   WAKUP   *STI
      JMP   TRNS3   *T3
*
PATH5:SKG   9BFF    *G.T. T4
      JMP   TRNS4   *T4
      SKG   CFFF    *G.T. E7
      JMP   SENDE   *E7
      JMP   SNDL2   *L2

*THIS SEQUENCE IS USED TO EXAMINE INSTRUCTIONS IN THE
*SCHEDULING PROCESSOR (I UNIT) WHICH WOULD ASSIGN
*INSTRUCTIONS TO OTHER PROCESSORS FOR EXECUTION (E UNIT)
*OR LOAD, STORE, MODIFY, OR TRANSFER (LSMT UNIT).
```

```
   PACE
OP-CODE GROUPS

HALT    00XY
E5      04XY
        10XX
T2      14XX
        18XX
E2      1CXX
        30BY
T1      4XXY
E1      5YYY
        74XY
STI     70XY
T3      8XXX
E3      88XX
        90XX
T4      94XX
        98XX
E7      9CXX
L2      A0XX
E6      A4XY
        AC??
L3      B0YX
E4      B8XX
L1      C0XX
        D0XX
E2      EXXX
        FXXX

E = EXECUTE

L = LOAD/STORE

T = TRANSFER/MODIFY
```

Figure 19

PACE Instruction Decoding Sequence

tigation proceeded on this basis. The suggested configuration and use of this system will be the last topic for discussion.

Configuration and Use of the Multi-microprocessor System

The multi-microprocessor system should be a generalized design adaptable to dedicated tasks depending on the application program. Two broad system classes would be development systems and production systems. The former would need additional capabilities for editing and manipulating memory while the latter would be more specialized according to the application. The associated equipment would be determined by user requirements, but both systems would have the same basic design.

This generalized system should have two or more processors with a common read/write memory and a private memory for each [12]. The private memories would be read/write memory for a development system and read only memory for a production system. Program tasks with synchronization primitives would be loaded into the private memories by the modified assembler. Processors would communicate using "mailboxes" (I/O ports) which would indicate messages in common memory [2,4]. See Appendix A. The individual processors would not need to be highly specialized unless this proved beneficial to the particular application.

Adapting the methods used here to any particular microprocessor language should not be difficult for someone who understands the subject machine, its op-codes, its instruction set, and the set of modifications made here. Although much work remains to be done, systems of this type are both feasible and useful. They can be developed relatively inexpensively for either research or commercial application.

BIBLIOGRAPHY

[1] M. J. Gonzales and C. V. Ramamoorthy, "Program Suitability for Parallel Processing," IEEE Trans. Computers, Vol. C-20, June 1971, pp. 647-654.

[2] P. Gebler, "Linking Microprocessors to Increase System Throughput," Electronic Engineering, Jan 1977, pp. 52-56.

[3] R. A. Perrin, "High Level Languages and the Microprocessor," Electronic Engineering, May 1977, pp. 65-67.

[4] A. J. Weissberger, "Analysis of Multiple Microprocessor System Architectures," Computer Design, June 1977, pp. 151-163.

[5] W. L. Spetz, "Microprocessor Networks," IEEE Computer, July 1977, pp. 64-70.

[6] K. Rozsa, "Multiprocessing Boosts Microcomputer Power Dramatically," Electronic Design, Vol. 6, Mar 15, 1978, pp. 72-75.

[7] T. Doone, "Microcomputer Multiprocessing Increases Throughput," Digital Design, May 1978, pp. 102-110.

[8] "Advanced Software Systems Design Course," (Editor's Tutorial), Electronic Design News, Oct 20, 1979, pp. 294-336.

[9] Y. P. Chien, "Multitasking Executive Simplifies Real Time Microprocessor System Design," Computer Design, Jan 1980, pp. 109-117.

[10] Lorin, "Moving a Single Processor System to its Limit," Parallelism in Hardware and Software: Real and Apparent Concurrency, (Englewood Cliffs, N. J: Prentice-Hall, 1972)

[11] M. J. Gonzalez and C. V. Ramamoorthy, "Survey of Techniques for Recognizing Parallel Processable Streams in Computer Programs," Fall Joint COMPCON 1969, AFIPS, Vol. 35.

[12] A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. Electronic Computers, Vol. 15, Oct 1966, pp. 757-763.

[13] C. V. Ramamoorthy, "Analysis of Graphs by Connectivity Considerations," ACM Journal, Vol. 13, April 1966, pp.211-222.

[14] "The FORTRAN Parallel Task Recognizer," Final NASA Report, Grant NGR 44-012-144, May 1970.

[15] T. F. Fox, Hon F. Li, and C. V. Ramamoorthy, "Scheduling Parallel Processable Tasks for a Uniprocessor," IEEE Trans. Computers, Vol. C-25, May 1976, pp. 485-495.

APPENDICES

APPENDIX A

SUGGESTED INTERPROCESSOR COMMUNICATIONS

This is a brief discussion of why interprocessor communication is necessary, how it could be accomplished, and how it could affect machine design. The two main reasons for communication between processors are resolution of conflicts between common resources such as shared memory and implementing task scheduling. Once the processors start executing a partition (set of tasks) they must follow a plan for transitioning to subsequent tasks in a predetermined fashion. Scheduling hueristics have been developed by Ramamoorthy [15]. These are probably applicable, but that discussion is beyond the scope of this thesis. The task execution will also generate resource conflicts that should be resolved in a systematic way. Djkstra, Knuth, and Coffman have developed efficient algorithms for scheduling shared resources [11]. No matter what system is used there must be a means to communicate between the processors.

It would probably be desirable to use the I/O ports of each processor as "mailboxes" [2,4]. This would mean that I/O would need to be accomplished by memory mapping to leave the ports free. Therefore, the interprocessor communication would have priority over I/O. The I/O could be designed to work through each processor's private memory, so the shared memory would not be involved either.

Interprocessor communication through mailbox messages has been used in systems such as MULTICS and also in smaller machines. It is an easy way to quickly indicate that one processor has a message for another and imply the degree of urgency. The notification requires only a byte or word in the form of an address on the I/O port. The message itself may be much longer as it can be stored in the common memory. The message notification may be accompanied by an interrupt signal if it is

important enough to deserve immediate attention. Otherwise, the receiving processor may be set up to check for messages at the end of a task or timed to check at specified intervals. If many messages are required, the mailbox may contain only a notification, handled on a schedule, that points to a part of the common memory where the messages are stored and prioritized. In this way the receiving processor can handle the messages as its schedule permits. But it will be able to accept a larger number of messages than would be possible if it waited to handle each individual message in the mailbox as it arrived. The way these messages are used would determine the system design to some extent.

After the program partitions have been scheduled and loaded with the proper pointers, the partitions would execute and point when done to the next partition. This could be done at least two ways. That is, the completed partition could return to an operating system or simply transfer control directly to the next process. This design decision would depend on the desired level of sophistication. By pointing to the next process directly it would seem possible to execute faster with less overhead. The proper synchronization primitives could be added to each partition by the loader, so that the task could not start until it was allowed to. Each partition would set its successor's primitives when finished. On the other hand, this function could be performed by the operating system by updating a table which would be checked by each partition before starting. If the process was not allowed to start, the operating system could retain control for more flexibility rather than simply idling the delayed processor. But in a non-dynamic scheduling

situation for a dedicated system, this degree of sophistication is probably not necessary. Tasks could communicate directly with each other with little overhead.

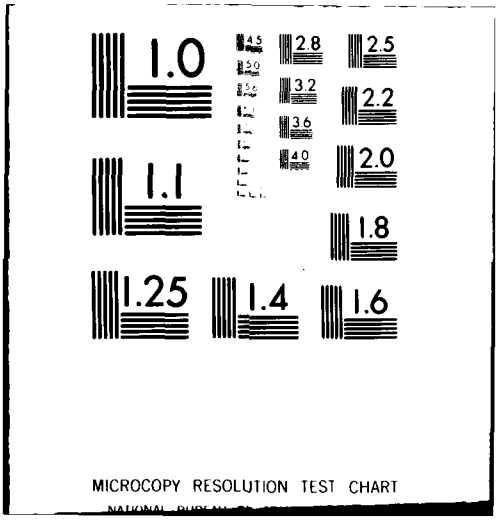
APPENDIX B

ASSEMLBER MODIFICATIONS LISTING

AD-A092 216 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH F/G 9/2
CONSIDERATIONS FOR AN ASSEMBLER SCHEDULED MULTI-MICROPROCESSOR --ETC(U)
UNCLASSIFIED AUG 80 R L STEWART
AFIT-CI-80-41T NL

2 OF 2
AD A
092216

END
DATE FILMED
1-81
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART

```
*ID P2411SC
*I P2411.46
C VERSION 2  15 APR 80 - PERFORMS SUITABILITY ANALYSIS
C                        FOR PARALLEL PROCESSING AND
C                        JUMP ANALYSIS (BOTH FOR INTEL
C                        8080 SOURCE CODE ONLY).
*I P2411.65
      COMMON/LOOP/LPMAX,LP(100),LPNO,LOOES
      COMMON/INST/JPX(1000,2),JP(200,4),ISTOP,MSTOPD,LPPG
*I P2411.71
      DIMENSION LP(100),JPOES(200)
*I P2411.110
      DATA JP,LP,JPOES/800*0,100*0,200*0/
      DATA JPX /2000*0/
*I P2411.129
      LINENM=0
*D P2411.160
      GO TO 500
*D P2411.167
      GO TO 500
*D P2411.173
      GO TO 500
*D P2411.180
      GO TO 500
*I P2411.114
C
C     ZERO VARIABLES FOR SUITABILITY EQUATION
C
C     SUITABILITY FACTOR
      SF=0.0
C     INPUT / OUTPUT INSTRUCTIONS
      NIO=0
C     ARITHMETIC / LOGICAL INSTRUCTIONS
      NAL=0
```

```
C     CALLS
      NC=0
C     JUMPS BACKWARDS (LOOPS)
      NB=0
C     JUMPS FORWARD (NOT RELATED TO A LOOP)
      NF=0
C     EXECUTABLE INSTRUCTIONS (NX ALSO USED TO DETERMINE NL)
      NX=1
C     INSTRUCTIONS WITHIN LOOPS
      NL=0
C
C     INITIALIZE JUMP TABLE INDEX, NJ
      NJ=1
C
C     JUMP TABLE CONTAINS: (1)JUMP ADDRESS, (2)DESTINATION
C     ADDRESS, (3)JUMP TYPE, AND (4)INDEX (NJ) OF
C     ASSOCIATED LOOP
C
C     JUMP TYPES ARE:                    LOOP CLASSES ARE:
C         1  CONDITIONAL BACKWARD            1  SIMPLE
C         2  UNCONDITIONAL BACKWARD          2  INTERMEDIATE
C         6  CONDITIONAL FORWARD             3  COMPLEX
C         8  UNCONDITIONAL FORWARD
C
C
C     WRITE HEADER FOR OUTPUT PAGE
C
      WRITE (5),6034)
 6034 FORMAT (1H1,11X,"---INTEL 8080 JUMP ANALYSIS--- VER "
     1   "2.00, 15 APR 80 --- "/)
*I P2411.147
      CALL SMSRCH(LABEL,KPTR)
      JDES(KPTR)=NX+1
```

```
*I P2411.184
      NC=NC+1
*D P2411.197
   14 GO TO(554,15,16,19,20,19,20,19,20,19,21,21,21,1),ICODE
*I P2411.207
      CALL SMSRCH(LABEL,KPTR)
      JPOES(KPTR)=NX+1
*I P2411.217
C     CHECK SUITABILITY FOR PARALLEL PROCESS BY DETERMINING
C     INSTRUCTION TYPES AND USING A FORMULA TO CHECK.
C
C     ARITHMETIC OR LOGICAL INSTRUCTION?
  5.1 IF(K.GT.2) GO TO 519
      IF(ICODE.EQ.1.OR.ICODE.EQ.119) GO TO 551
      IF(ICODE.LT.27.AND.(MOD(ICODE,8).EQ.2)) GO TO 55
      IF(ICODE.GT.191) GO TO 55
C
C     COUNT ARITHMETIC OR LOGICAL INSTRUCTION
C
  519 NAL=NAL+1
      GO TO 55
C
C     CALL OR JUMP?
C
  51 IF(K.LE.5) GO TO 54
      IF(MOD(ICODE,8).LT.4) GO TO 520
C     CONDITIONAL CALL?
      IF(MOD(ICODE,8).NE.5) GO TO 512
      NC=NC+1
      GO TO 515
  512 NF=NF+1
  515 GO TO 55
  520 IF(ICODE.LT.194) GO TO 55
C
C     JUMPS - CHECK TYPE & BUILD TABLE
```

```
C     FIRST FIND IF DEST IS DEFINED (I.E. FWD OR BACK)
C
      CALL SMSRCH(GP1,KDEST)
C
      IF(ICODE.NE.135) GO TO 527
C     UNCONDITIONAL. IF FWD, DON'T KNOW DEST ADDRESS YET.
      IF(LOCS(KDEST).EQ.-1) GO TO 525
C     DESTINATION INSTRUCTION NO. OF JUMP BACK
      JP(NJ,2)=JPDES(KDEST)
C     JUMP CLASS
      JP(NJ,3)=2
      JP(NJ,4)=NJ
      GO TO 529
C     UNCONDITIONAL JUMP FORWARD
  525 JP(NJ,2)=-KDEST
      JP(NJ,3)=3
      GO TO 530
C     CONDITIONAL. IF FWD, DON'T KNOW DEST ADDRESS YET.
  527 IF(LOCS(KDEST).EQ.-1) GO TO 528
C     DESTINATION INSTRUCTION NO. OF JUMP BACK
      JP(NJ,2)=JPDES(KDEST)
C     JUMP CLASS
      JP(NJ,3)=1
      JP(NJ,4)=NJ
      GO TO 529
C     JUMP FORWARD
  528 JP(NJ,2)=-KDEST
      JP(NJ,3)=4
      GO TO 530
C
C     COUNT A LOOP AND INCREMENT LOOP TABLE POINTER
C
  529 NP=NP+1
C
```

```
C     SAVE JUMP TABLE INDEX OF LOOP
C
      LP(NB)=NJ
C
C     JUMP TABLE OVERFLOW
C
  530 CONTINUE
      IF (NB.GT.100) WRITE (50,9991)
 9991 FORMAT (1X,"+++WARNING. ARRAY LP OVERFILLED. "
     1   "MORE THAN 100 LOOPS."/)
      IF(NJ.GT.200) WRITE (50,9996)
 9996 FORMAT(1X"+++WARNING. ARRAY JP OVERFILLED. MORE THAN"
     1   " 200 JUMPS."/)
C
C     JUMP ADDRESS
C
      JP(NJ,1)=NX+1
C
C     INCREMENT JUMP TABLE POINTER
C
      JPX (LINENM,1) = NJ
      NJ=NJ+1
      GO TO 550
C
C     I/O?
C
  540 IF(K.NE.3) GO TO 550
      IF(ICODE.EQ.211.OR.ICODE.EQ.219) NIO=NIO+1
      IF(MOD(ICODE,8).EQ.6) GO TO 540
C
C     COUNT EXECUTABLE INSTRUCTIONS
C
  550 NX=NX+1
      JPX(LINENM,2) = NX
```

```
      GO TO 1
C
C     CHECK LOOP CONSTRUCTS AT END OF PASS ONE
C
C     SAVE HOW MANY JUMPS (NJMAX)
C
  554 CONTINUE
      NJMAX=NJ-1
C
C     SAVE NUMBER OF LOOPS
C
      LPMAX = NB
C
C     FIND NX OF FWD JUMP DESTINATIONS TO USE
C     IN CALCULATIONS FOR NL LATER
C
C     CHECK ALL JUMPS
C
      DO 557 JC=1,NJMAX
C     ALREADY KNOW NX
      IF(JF(JC,2).GT.0) GO TO 557
C     IF NOT, CHECK ALL SYMBOLS
      DO 555 JS=1,SMBPTR
C     IS THIS JUMP ASSOC WITH THIS SYMBOL?
      IF(JF(JC,2).EQ.-JS) JF(JC,2)=JPDES(JS)
  555 CONTINUE
  557 CONTINUE
C
C     MAIN LOOP OF THIS PORTION INDEXES JUMP BEING CHECKED
C
      DO 590 MC=1,NJMAX
      MJ=NJ-MC
C     HAS THIS JUMP BEEN ASSOCIATED WITH A LOOP?
      IF(JF(MJ,4).GT.0.AND.JF(MJ,3).GT.2) GO TO 59
```

```
C      BRANCH ACCORDING TO JUMP TYPE
       IF(JP(MJ,3).GT.0.AND.JP(MJ,3).LT.3) GO TO 562
       IF(JP(MJ,3).EQ.6) GO TO 576
       IF(JP(MJ,3).EQ.8) GO TO 576
C     IF IT WAS NOT ONE OF THE ABOVE, SHOW AN ERROR MSG.
       WRITE(50,9993) MJ
 9993 FORMAT(1X,"***WARNING. UNCLASSIFIED JUMP #",I4,"./)
C      DETERMINE LOOP CLASS 1, 2, OR 3
C
C      THIS LOOP CHECKS FORWARD JUMPS (NF1) TO ASSOCIATE
C      THEM WITH THE BACKWARD JUMP (MJ). TO BE ASSOCIATED,
C      THE FORWARD JUMP MUST START IN A LOOP.
C
  562 CONTINUE
C      INITIALIZE ENTRY (KFE) AND EXIT (KFO) FLAGS TO NONE
       KFE=0
       KFO=0
       MF1=MJ-1
       DO 574 NC1=1,MF1
       NF1=MJ-NC1
C      IS IT A FWD JUMP?
       IF(JP(NF1,3).NE.6.AND.JP(NF1,3).NE.8) GO TO 574
C      YES. DOES IT ORIGINATE IN THE LOOP?
       IF(JP(NF1,1).GE.JP(MJ,2).AND.JP(NF1,1).LT.JP(MJ,1))
      1 GO TO 564
C
C      NO. DOES IT JUMP IN?
C
       IF(JP(NF1,2).LT.JP(MJ,2).OR.JP(NF1,2).GT.JP(MJ,1))
      1 GO TO 574
       CALL FINDLP(MJ)
       WRITE (50,9903) NF1,MJ,LPNO
       GO TO 574
C
```

```
C     DOES ORIGINATE IN THE LOOP. DOES IT JUMP OUT?
C
  564 CONTINUE
      IF(JP(NF1,2).GT.JP(MJ,1)) GO TO 565
C     NOTE INSIGNIFICANT INTERIOR FWD JUMP.
      CALL FINDLP(MJ)
      IF (JP(MJ,3).EQ.1) WRITE (50,9995) NF1,MJ,LP,C
      GO TO 572
C     YES. JUMP OUT. ENTRY IF CONDITIONAL.
  565 CONTINUE
      IF(JP(NF1,3).EQ.6) KFE=1
C     EXIT MAY EXIST SINCE THERE IS A JUMP OUT.
      KFO=1
C
C     CHECK ALL PREVIOUS FWD JUMPS TO DETERMINE EXIT.
C
      NCA1=NF1-1
      DO 570 NC16=1,NCA1
      N6=NF1-NC16
C     IS IT FWD?
      IF(JP(N6,3).NE.6.AND.JP(N6,3).NE.8) GO TO 57
C     INSIDE JUMP AROUND?
      IF(JP(N6,2).LT.JP(MJ,2)) GO TO 57
      IF(JP(N6,1).GT.JP(NF1,1).OR.JP(N6,2).LT.JP(NF1,1)
     1  .OR.JP(N6,2).GT.JP(MJ,1).OR.JP(MJ,2).GT.JP(N6,1))
     2  GO TO 57
      IF(JP(N6,4).GT.0) WRITE(50,9997) N6,IABS(JP(N6,4))
      JP(N6,4)=JP(MJ,4)
 9997 FORMAT(1X,"NOTE. COND FWD JUMP"I4" ENTERS BACK JUMP"
     1   I4" (LOOP "14")  FROM OUTSIDE ITS RANGE."/)
C     CURRENT JUMP TYPE FROM DO 57+?
      IF(JP(NF1,3).EQ.8) GO TO 566
C     TYPE 8. IS JUMP AROUND A TYPE 8?
      IF(JP(N6,3).EQ.8) GO TO 57
```

```
 9902 FORMAT(1X"*** POSSIBLE ERROR. NO ENTRY TO LOOP # "I4
     1  " (BACK JUMP "I4" ) UNLESS BY RETURN INSTRUCTION "
     2  "FROM JUMP OUT."/)
      KFE=1
C
C     COMPLEX CLASS 3 LOOP?
C
      IF(JP(MJ,3).NE.1) JP(MJ,3)=3
      GO TO 570
  566 CONTINUE
C
C     TYPE 6. IS JUMP AROUND A TYPE 6?
C
      IF(JP(N6,3).EQ.6) KFO=0
  570 CONTINUE
      CALL FINDLP(MJ)
      IF(JP(MJ,3).EQ.2.AND.KFE.NE.1) WRITE (5,9992) LPNO,MJ
      KFE=0
  572 CONTINUE
      IF(JP(NF1,4).NE.0.AND.JP(NF1,4).NE.JP(MJ,4))
     1 WRITE (5,9997) NF1,IABS(JP(NF1,4))
      JP(NF1,4)=JP(MJ,4)
  574 CONTINUE
      CALL FINDLP(MJ)
      IF(JP(MJ,3).EQ.2.AND.KFO.NE.1) WRITE (5,9995) MJ,LPNO
      GO TO 590
 9995 FORMAT(1X,"NOTE. COND FWD JUMP "I4" AND BACK JUMP "
     1  I4" (LOOP"I4" ), BUT IS NOT SIGNIFICANT TO ITS "
     2  "STRUCTURAL CLASSIFICATION."/)
 9997 FORMAT(1X,"NOTE. FWD JUMP"I4" ALSO ASSOC WITH JUMP"
     1  I4"."/)
C
C     THIS LOOP CHECKS ALL SUBSEQUENT LOOPS (NF1) AGAINST
C     THE FWD JUMP (MJ) FOUND IN DO 570 TO SEE IF MJ JUMPS
```

```
C      AROUND ANY OF THE OTHER LOOPS (NF8).
C
C
  576 CONTINUE
      NC8L=NJMAX-MJ
      DO 577 NC8=1,NC8L
      NF8=NJ-NC8
C     IS NF8 A BACKWARD JUMP?
      IF(JP(NF8,3).GT.3) GO TO 577
C     YES. DOES MJ JUMP AROUND LOOP NF8?
      IF(JP(MJ,2).LE.JP(NF8,1).OR.JP(MJ,1).GT.JP(NF8,2))
     1   GO TO 577
C
C     DID IT JUMP AROUND A FWD JUMP ALSO?
C
      IF (JP(MJ,4).LT.0.AND.JP(-JP(MJ,4),3).GT.4)
     1   WRITE (60,9990) MJ, -JP(MJ,4)
      CALL FIND_LP(NF8)
      WRITE(60,9994) MJ,NF8,LPNO
 9994 FORMAT(1X,"NOTE. FWD JUMP"I4" AROUND BACK JUMP"
     1  I4" (LOOP"I4" )"/)
C     ASSOCIATE WITH -LOOP
      JP(MJ,4)=-NF8
  577 CONTINUE
      NFU=MJ-1
C     THIS LOOP CHECKS ONLY PREVIOUS FWD JUMPS (NCU) AGAINST
C     THE FWD JUMP (MJ) TO SEE IF THERE IS A JUMP AROUND MJ.
      DO 584 NCF=1,NFU
      NCU=MJ-NCF
C     IS NCU A FWD JUMP?
      IF(JP(NCU,3).LT.4) GO TO 584
C     YES. DOES NCU JUMP AROUND MJ?
      IF(JP(NCU,2).LE.JP(MJ,1)) GO TO 584
C     YES.
```

```
C
C      ALREADY ASSOC WITH OTHER FWD JUMPS?
C
       IF (JP(NCU,4).LT.0.AND.(JP(NCU,4)).NE.-MJ)
     1   NCU,-JP(NCU,4)
     1   WRITE (50,9999) NCU,-JP(NCU,4)
C      ASSOC JUMP NCU WITH -MJ TO SHOW JUMP AROUND.
 9999 FORMAT (1X,"NOTE. JUMP "I4" ALSO JUMPED AROUND "
     1  "JUMP"I4"."/)
       JP(NCU,4)=-MJ
C      NCU CONDITIONAL?
       IF(JP(NCU,3).EQ.0) GO TO 584
       WRITE(50,9993) NCU,NFA
  584 CONTINUE
 9993 FORMAT (1X,"*** POSSIBLE ERROR. UNCOND FWD JUMP "
     1   I4" AROUND FWD JUMP "I4"."/)
C      IF FWD JUMP (MJ) IS NOT ASSOC, ASSOC IT WITH ITSELF
       IF(JP(MJ,4).EQ.0) JP(MJ,4)=-MJ
 9001 FORMAT (1X,"*** POSSIBLE ERROR. NO EXIT FROM BACK "
     1   "JUMP"I4" (LOOP"I4") UNLESS BY RETURN INSTRUCTION "
     2   "OR OVERLAPPED LOOP."/)
C      END MAIN LOOP TO CHECK JUMPS
   , CONTINUE
C
C      THIS LOOP CHECKS THE FINAL JUMP TABLE TO FIND WHICH
C      LOOPS ARE NESTED AND TO SEE IF ANY ARE OVERLAPPED
C      WHICH WOULD CAUSE ERRORS IN THE JUMP ANALYSIS.
       LCDES=LPMAX
C      FLAG OUTERMOST LOOP FOR LISTING
       LP(LCDES) = -LP(LCDES)
C
       DO 588 KCN=1,LPMAX
C      CHECK THROUGH LOOP TABLE BACKWARDS
       KC=(LPMAX+1)-KCN
```

```
C     IF FIRST LOOP IN PROGRAM, NO NEED TO CHECK NEST.
      IF(KC.EQ.1) GO TO 587
C     PREVIOUS LOOP NESTED IN PRESENT LOOP (KC)?
      IF(JP((IABS(LP(KC))),2).LE.JP(LP(KC-1),2)) GO TO 586
      IF(KC.EQ.LCOES) GO TO 585
C     PREVIOUS LOOP NESTED IN OUTERMOST LOOP (LCOES)?
      IF(JP((IABS(LP(LCOES))),2).LE.JP(LP(KC-1),2))
     1   WRITE (60,9982) KC-1,LCOES
C     PREVIOUS LOOP OVERLAPPED WITH PRESENT LOOP (KC)?
  585 IF(JP(LP(KC-1),1).GE.JP(LP(KC),2).AND
     1  .JP(LP(KC),2).GT.JP(LP(KC-1),2))
     2  WRITE (60,9981) KC-1,KC
 9981 FORMAT (1X,"+++WARNING. LOOP"I4" OVERLAPPED WITH LOOP"
     1  I4". SF VALUE MAY NOT BE MEANINGFUL. LFPS MAY NOT"
     2  " BE ACCURATE."/)
      GO TO 587
  586 WRITE (60,9982) KC-1,KC
 9982 FORMAT (1X,"NOTE. LOOP"I4" NESTED IN LOOP"I4/)
C     WHAT IS LCOES?
  587 IF(JP(LP(KC),2).GE.JP((IABS(LP(LCOES))),2)) GO TO 588
C     NOTE OUTERMOST LOOP IN TABLE
      LCOES = KC
      IF (LCOES.EQ.LPMAX) GO TO 588
      LP(LCOES) = -LP(LCOES)
  588 CONTINUE
C
C
C     THIS LOOP CHECKS THE FINAL JUMP TABLE TO
C     FIND THE TOTAL NUMBER OF FORWARD JUMPS
C     NOT ASSOCIATED WITH LOOPS
C
      DO 591 ICJ=1,NJMAX
      IF (JP(ICJ,4).LE.0) NF=NF+1
  591 CONTINUE
```

```
C     COUNT INSTRUCTIONS IN LOOPS
      INSTL=0
      DO 595 NCNT=1,LPMAX
      IF(LP(NCNT).GE.0) GO TO 594
      INSTL = JP(IABS(LP(NCNT)),1) - JP(IABS(LP(NCNT)),2)
      NL=NL+INSTL+1
      GO TO 595
  594 CONTINUE
      NB = NB - 1
  595 CONTINUE
      IC=FLOAT(NIC)
      AL=FLOAT(NAL)
      UC=FLOAT(NC)
      BJ=FLOAT(NB)
      FJ=FLOAT(NF)
      EX=FLOAT(NX)
      PL=FLOAT(NL)
      IF((NX-NL).LE.0) GO TO 5952
      SF=((IC+AL+UC+BJ-FJ)/(EX-PL))-(PL/EX)
      WRITE (50,9000) SF
 9000 FORMAT (" SUITABILITY FACTOR FOR PARALLEL PROCESSING"
     1  " IS "F7.2". ANY VALUE GREATER THAN 0.3 IS FAVOR-"
     2  "ABLE."/)
      GO TO 596
 5952 WRITE (50,9010)
 9010 FORMAT (1X"*** LOOP STRUCTURE IS NOT SUITABLE FOR "
     1   "PARALLEL PROCESSING AT THIS LEVEL."/)
  596 CONTINUE
C
C     PRINT THE JUMP TABLE
C
 5666 WRITE (50,5999)
 5999 FORMAT (1X,T53,"JUMP TABLE"//
     1 1X,T46,"NJMAX"2X"NIC"2X"NAL"2X"NC"3X"NJ"3X"NF"3X
```

```
     2 "NL"3x"NX")
       WRITE (50,6000) NJMAX,NIO,NAL,NC,NB,NF,NL,NX
 6000 FORMAT (1X,T46,8(I4,X)/)
       WRITE (50,6005)
 6005 FORMAT (1X,T45"LOOP JUMP",1X,"FROM",2X,"TO",2X,"TYPE"
     1  1X"ASSOC")
       JNX=0
       DO 7000 ID=1,NJMAX
       WRITE (50,6010) ID,(JP(ID,IV),IV=1,4)
 6010 FORMAT (1X,T50,I3,X,4(I4,X)/)
       IF(ID.EQ.JP(ID,4)) JNX=JNX+1
       IF(ID.EQ.JP(ID,4)) WRITE (50,6015) JNX
 6015 FORMAT (1X,T45,I3,2X,25H*************************)
 7000 CONTINUE
C
C      PRINT THE JUMP ANALYSIS
C
       WRITE(50,9400)
 9400 FORMAT (1X,T47,"JUMP ANALYSIS"/)
C      INDEX THROUGH JUMP TABLE
       DO 597 IC=1,NJMAX
       IF(JP(IC,3).LT.6) GO TO 597
C      CHECK FOR ANY CIRCUMVENTED INSTRUCTIONS (UNCOND JUMP)
       IF(JP(IC,3).NE.8) GO TO 5964
C      LOOK FOR PREVIOUS FORWARD JUMP TO INSTRUCTION
C      AFTER UNCONDITIONAL JUMP
       ICL=IC-1
       DO 5962 IDC = 1, ICL
       IDCN = IC - IDC
       IF(JP(IDCN,2).EQ.(JP(IC,1)+1)) GO TO 5964
 5962 CONTINUE
C      LOOK FOR LOOP BACK TO INSTRUCTION AFTER UNCOND JUMP
       ICN=IC+1
       DO 5963 IDC=ICN,NJMAX
```

```
      IF(JP(IDC,3).GT.3) GO TO 5963
      IF(JP(IDC,2).EQ.(JP(IC,1)+1)) GO TO 5964
 5963 CONTINUE
C     IF NO PATH WAS FOUND, WRITE ERROR MESSAGE
      NOPATH = JP(IC,1)+1
      WRITE (50,9493) NOPATH
 9493 FORMAT(1X,T18,"*** POSSIBLE ERROR. NO PATH TO "
     1   "INSTRUCTION"I4"UNLESS BY CALL."/)
 5964 IF(JP(IC,4).NE.0.AND.JP(IC,4).NE.-IC) GO TO 597
      WRITE (50,9490) IC,JP(IC,1)
 9490 FORMAT (1X,T30,"NOTE. FWD JUMP "I3", EX INSTR NO."I4
     1   " IS NOT ASSOCIATED WITH ANY LOOP."/)
  597 CONTINUE
      WRITE(50,9500)
 9500 FORMAT(9X,"LOOPS IN THIS PROGRAM"/
     1   1X," LOOP #",5X,"FROM",6X," TO ",6X,"CLASS"/)
      DO 599 IL=1,LP1AX
      WRITE(50,9520) IL,JP(IABS(LP(IL)),1),JP(IABS(LP(IL))
     1  ,2),JP(IABS(LP(IL)),3)
 9520 FORMAT(3X,I4.4,6X,I4.4,6X,I4.4,6X,I3)
      WRITE(50,9530) IL
 9530 FORMAT(6X,T50,"FORWARD JUMPS ASSOCIATED WITH "
     1  "LOOP ",I3/)
      KFF=0
      DO 598 IC=1,NJMAX
      IF(JP(IC,4).NE.LP(IL)) GO TO 598
      IF(JP(IC,3).GE.1.AND.JP(IC,3).LE.3) GO TO 598
      WRITE(50,9496) IC,JP(IC,1)
 9496 FORMAT(1X,T50,"JUMP"I3" EX INSTR NO. "I4,/)
      KFF=1
  598 CONTINUE
      IF(KFF.EQ.0) WRITE(50,9497)
 9497 FORMAT (1X,T62,"NONE"/)
  599 CONTINUE
```

```
      GO TO 22
*I P2411.238
C     INITIALIZE TO SHOW LOOP FREE PROGRAM GRAPH (LFPG)
      LFPG=0
      MSTOPN = NX
C     ANY LOOPS?
      IF(LPMAX.EQ.0) GO TO 23
C     YES. RESTORE LOOP TABLE ENTRY
      LP(LODES) = -LP(LODES)
C     SET FLAG TO STOP NUMBERS AT DESTINATION
      MSTOPN = JP(LP(LODES),2)
C     SET FLAG TO STOP . AT JUMP
      MSTOPD = JP(LP(LODES),1)
*I P2411.818
     1 31MVER 2.00, 19 APR 80 ---   PAGE ,13//)
*I P2411.819
      SUBROUTINE FINDLP(I)
C
C     THIS ROUTINE FINDS THE LOOP NUMBER (LPNO) OF
C     A BACK JUMP (I) FOR REFERENCE PURPOSE IN DISASSEMBLING
C
      COMMON /LOOP/ LPMAX,LP(100),LPNO,LODES
      LPNO=0
      DO 100 LCNT=1,LPMAX
      IF(I.EQ.LP(LCNT)) GO TO 300
  100 CONTINUE
      IF(LPNO.EQ.0) WRITE(50,9000) I
 9000 FORMAT (1X,"+++WARNING. LOOP NO. FOR BACK JUMP"I4
     1    " NOT FOUND."/)
  300 LPNO=LCNT
      RETURN
      END
*I P2411.721
      COMMON/INST/JPX(1000,2),JP(200,4),MSTOPN,MSTOPD,LFPG
```

```
      COMMON/LOOP/LPMAX,LP(300),LPMAX,LOOPS
*I P2-11,723
      LPX = LPMAX - 1
*D P2-11,740,P2-11,741
C
C     SHOW NODES OF LOOP FREE PROGRAM GRAPH
C               (LPFG) ON LISTING
C
C     COMMENT?
*     WRITE (6,6021) JPX(LINENM,2)
 6021 FORMAT (1LX,"EX JO, ="I4/)
      IF(JPX(LINENM,2).EQ.0) GO TO 300
C     COME TO AN OUTSIDE LOOP YET?
      IF(JPX(LINENM,2).LT.MSTOPM) GO TO 200
C     NO. DONE?
      IF(JPX(LINENM,2).EQ.MX) GO TO 200
C     NO. IN AN OUTSIDE LOOP?
      IF(JPX(LINENM,2).LT.MSTOPL) GO TO 200
C     LAST LOOP?
      IF(LOOPS.EQ.LPMAX) LP(LOOPS) = -LP(LOOPS)
      IF(LOOPS.EQ.LPMAX) GO TO 200
C     AT JUMP FOR AN OUTSIDE LOOP? RESET.
      DO 50 LC = LOOPS,LPX
C     FOUND NEXT OUTSIDE LOOP?
      IF(LP(LC+1).LT.0) GO TO 75
   50 CONTINUE
C     SET NEXT OUTSIDE LOOP.
   75 LOOPS = LC + 1
C     RESTORE LOOP TABLE AND RESET FLAGS
      LP(LOOPS) = -LP(LOOPS)
      MSTOPM = JP(LP(LOOPS),2)
      MSTOPL = JP(LP(LOOPS),1)
      GO TO 200
C     SUB NODE LINE
```

```
  1.. WRITE(50,12) LINEN ,PCTR,((OP(J,I),J=1,3),I=1,3)
     1     ,(JPX(LINENM,K),K=1,2),LINE
           GO TO 400
C          NODE LINE
  200 LPPG=LPPG+1
           WRITE(50,11) LINENM,PCTR,((OP(J,I),J=1,2),I=1,3)
     1     ,JPX(LINENM,1),LPPG,JPX(LINENM,2),LINE
           GO TO 400
C          COMMENT LINE
  300 WRITE(50,3) LINENM,PCTR,((OP(J,I),J=1,3),I=1,3)
     1     ,(JPX(LINENM,K),K=1,2),LINE
  400 IF(PCNTR.LT.3) RETURN
*0 P2=11,7=5
    3 FORMAT (1H ,I5,=X,A41,2X,3(2A1,1X),I4,=,1 X,A=,=,3X,5A41)
*0 P2=11,7dd
     1 31HVER 2.0?, 10 APR  6 ---   PAGE ,I3//
     2 T4,"LINE",T12,"P C",T17,"MACH CODE",T24,"JUMP  LF"
     3 "PG  EX OC"T=7,"LABEL"T15="INSTRUCTION"T30"OPERANDS"T=3
     4 "COMMENTS"//)
   11 FORMAT (1H ,I5,3A,=A1,2X,3(2A1,1X),I4,=,3X,I4,=,
     1  3X,I4,=),3X,30A1)
   12 FORMAT (1H ,I5,=X,A41,2X,3(2A1,1X),I4,=,"       . "
     1  2X,I4,=,3X,30A1)
*0 P2=11,993
C          SYMPTR -- ARRAY NUMBER OF LAST ENTRY
C                    IN SYMBOL TABLE
*0 P2611
```

APPENDIX C

PROGRAM SPECIFICATIONS OF THE

MODIFIED ASSEMBLER

This is a brief discussion of the size of the program and how much time it takes to run. This modified assembler requires approximately 34K to load in a CDC 6600 and 54K to execute with its associated system routines. The compile time is approximately 8.7 seconds, but it could be loaded from disk in a fraction of that time. The execution time depends on source program length to an extent, but mainly on source program complexity. Execution times of 0.645 central processor (CP) seconds to 9.1 CP seconds have been noted. These were for approximately 85 and 1200 program lines respectively including instructions and comments. However, a test program of approximately 120 lines that had many jumps took 8.2 CP seconds to execute. All these numbers depend also on the host assembler to some extent. The host used is a fairly sophisticated large program that can assemble either INTEL or MOTOROLA source code. To modify the assembler, arrays totaling approximately 3100 words were required to be added. The present version can accomodate source programs with up to 200 jumps, 100 loops, and 1000 lines of instructions and comments. These arrays can be easily adjusted to smaller or larger source programs.

APPENDIX D

INTEL 8080 OP-CODE GROUPS

For INTEL 8080 assembly language, checking op-codes to count instructions is easier than checking mnemonics. Mnemonics would require checking character by character. The result obtained would require checking against a table or some standard to decide which ones to count (NAL, NC, NIO, or NJ). But op-codes are made available in pass one by this assembler for checking assembler directives (pseudo op-codes). Since pseudo ops are of no concern for this analysis of instruction types, only valid instructions are checked.

Instruction op-codes are grouped in a way that allows easy instruction identification. Identification is made by using the last digit of the op-code and the INTEL instruction type. The eight INTEL instruction types (called K in the assembler) are based on the references made and instruction length. These are used to determine the instructions in each of three main op-code groups. These octal groups are conveniently divided as 0 to 177, 200 to 277, and 300 to 376. In the first group, all instructions for which K is three are arithmetic or logical, except if the last digit (modulo 8) is two, or if the op-code is 0 or 166. In the second group, all are arithmetic or logical. In the third group, if K equals four, the instructions are either jumps or calls. But if K equals three, they are I/O unless the last digit is six which indicates arithmetic or logical immediate instructions. This is why it is so easy to determine the type and count the numbers of each. If the op-code does not fall into one of these groups, it is simply counted as an executable instruction.